Homework 4 Due: October 14th @ 11:59PM (Gradescope)
Reminder: You must include a collaborator/resource (CR) statement for every problem.
3.1 Basic Definitions and Applications
Undirected Graphs

Undirected graph. $G = (V, E)$
- $V$ = nodes.
- $E$ = edges between pairs of nodes.
- Captures pairwise relationship between objects.
- Graph size parameters: $n = |V|$, $m = |E|$.

$V = \{ 1, 2, 3, 4, 5, 6, 7, 8 \}$
$E = \{ 1-2, 1-3, 2-3, 2-4, 2-5, 3-5, 3-7, 3-8, 4-5, 5-6 \}$
$n = 8$
$m = 11$
Transportation Networks (Planes, Trains, Highways etc...)
Web graph.

- Node: web page.
- Edge: hyperlink from one page to another.
9-11 Terrorist Network

**Social network graph.**

- **Node:** people.
- **Edge:** relationship between two people.

Ecological Food Web

Food web graph.

- **Node = species.**
- **Edge = from prey to predator.**

Graph Representation: Adjacency Matrix

**Adjacency matrix.** An n-by-n matrix with $A_{uv} = 1$ if $(u, v)$ is an edge.
- Two representations of each edge.
- Space proportional to $n^2$.
- Checking if $(u, v)$ is an edge takes $\Theta(1)$ time.
- Identifying all edges takes $\Theta(n^2)$ time.

```
1 2 3 4 5 6 7 8
1 0 1 1 0 0 0 0 0
2 1 0 1 1 1 0 0 0
3 1 1 0 0 1 0 1 1
4 0 1 0 0 1 0 0 0
5 0 1 1 1 0 1 0 0
6 0 0 0 0 1 0 0 0
7 0 0 1 0 0 0 0 1
8 0 0 1 0 0 0 1 0
```
Graph Representation: Adjacency List

**Adjacency list.** Node indexed array of lists.
- Two representations of each edge.
- Space proportional to $m + n$.
- Checking if $(u, v)$ is an edge takes $O(\text{deg}(u))$ time.
- Identifying all edges takes $\Theta(m + n)$ time.

**degree** = number of neighbors of $u$
Paths and Connectivity

Def. A path in an undirected graph $G = (V, E)$ is a sequence $P$ of nodes $v_1, v_2, ..., v_k$ with the property that each consecutive pair $v_i, v_{i+1}$ is joined by an edge in $E$.

Def. A path is **simple** if all nodes are distinct (e.g., 1,2,5,6 below)

Def. An undirected graph is **connected** if for every pair of nodes $u$ and $v$, there is a path between $u$ and $v$. 
Def. A **cycle** is a path \( v_1, v_2, ..., v_{k-1}, v_k \) in which \( v_1 = v_k, k > 2 \), and the first \( k-1 \) nodes are all distinct.

cycle \( C = 1-2-4-5-3-1 \)
**Def.** An undirected graph is a tree if it is connected and does not contain a cycle.

**Theorem.** Let $G$ be an undirected graph on $n$ nodes. Any two of the following statements imply the third.
- $G$ is connected.
- $G$ does not contain a cycle.
- $G$ has $n-1$ edges.
Rooted tree. Given a tree $T$, choose a root node $r$ and orient each edge away from $r$.

Importance. Models hierarchical structure.
**Binary Tree**

**Def.** A rooted tree in which each node has at most 2 children

**Def.** Height of a tree is the number of edges in the longest path from root to leaf.

![Binary Tree Diagram](image)

**Thm.** Number of nodes in binary tree of height $h$ is $n \leq 2^{h+1} - 1 = 2^0 + 2^1 + 2^2 + \cdots + 2^h$.

**Balanced Binary Tree.** Height $h = O(\log n)$
Let $G=(V,E)$ be a simple undirected graph. Which of the following claims are false?

A. If $|E| > n-1$ then $G$ contains a cycle

B. If $|E| = n-1$ then $G$ is a tree

C. If $|E| = \frac{n(n+1)}{2}$ then $G$ is connected

D. If $G$ is represented as an adjacency matrix then we can test whether or not a particular edge $(u,v)$ exists in time $O(1)$

E. We can store $G$ using $O(m+n)$ space using adjacency list representation.
Let $G=(V,E)$ be a simple undirected graph. Which of the following claims are false?

A. If $|E| > n-1$ then $G$ contains a cycle

B. If $|E| = n-1$ then $G$ is a tree

C. If $|E| = n(n+1)/2$ then $G$ is connected
   ($G=K_n$ is complete graph & contains all possible edges)

D. If $G$ is represented as an adjacency matrix then we can test whether or not a particular edge $(u,v)$ exists in time $O(1)$

E. We can store $G$ using $O(m+n)$ space using adjacency list representation.
3.2 Graph Traversal
Connectivity

s-t connectivity problem. Given two node s and t, is there a path between s and t?

s-t shortest path problem. Given two node s and t, what is the length of the shortest path between s and t?

Applications.
- Navigation (Google Maps).
- Maze traversal.
- Kevin Bacon number (or Erdős Number).
- Fewest number of hops in a communication network.
Breadth First Search: BFS(v, G=(V,E))

Input: Start node v, graph G (adjacency list) with n node V={1, ..., n}

Output: Array Level[u] = distance from v to u

For each node u in V
    Explored[u]=0
    Level[u] = ∞      // No v→u path found yet

Q.Enqueue(v)       // Queue Q (First In, First Out)

Level[v]=0

while(Q is not empty)
    u = Q.Dequeue()
    if (Explored[u] = 0)
        Explored[u]=1
        foreach node w in u.AdjList
            if (Level[w] = ∞)
                Q.Enqueue(w)
                Level[w] = Level[u]+1
    end if
Breadth First Search: \( \text{BFS}(v, G=(V, E)) \)

**Input:** Start node \( v \), graph \( G \) (adjacency list) with \( n \) node \( V=\{1,\ldots,n\} \)

**Output:** Array \( \text{Level}[u] = \) distance from \( v \) to \( u \)

For each node \( u \) in \( V 

- \( \text{Explored}[u]=0 \)
- \( \text{Level}[u] = \infty \) \hspace{1cm} // No \( v \rightarrow u \) path found yet

**Q. Enqueue** \( (v) \) \hspace{1cm} // Queue \( Q \) (First In, First Out)

\( \text{Level}[v]=0 \)

**while** (\( Q \) is not empty)

- \( u = \text{Q.Dequeue()} \)
- if (\( \text{Explored}[u] = 0 \))
  - \( \text{Explored}[u]=1 \)
  - foreach node \( w \) in \( u.\text{AdjList} \)
    - if (\( \text{Level}[w] = \infty \))
      - \( \text{Q.Enqueue}(w) \)
      - \( \text{Level}[w] = \text{Level}[u]+1 \)
    - end if

end if
Breadth First Search: \textbf{BFS}(v, G=(V,E))

**Input:** Start node \(v\), graph \(G\) (adjacency list) with \(n\) nodes \(V=\{1,\ldots,n\}\)

**Output:** Array \(\text{Level}[u] = \text{distance from } v \text{ to } u\)

For each node \(u\) in \(V\)
- \(\text{Explored}[u]=0\)
- \(\text{Level}[u] = \infty\) // No \(v\rightarrow u\) path found yet

\(\text{Q.Enqueue}(v)\) // Queue \(Q\) (First In, First Out)
- \(\text{Level}[v]=0\)

\(\text{while}(Q \text{ is not empty})\)
  - \(u = \text{Q.Dequeue}()\)
  - if (\(\text{Explored}[u] = 0\))
    - \(\text{Explored}[u]=1\)
    - foreach node \(w\) in \(u.\text{AdjList}\)
      - if (\(\text{Level}[w] = \infty\))
        - \(\text{Q.Enqueue}(w)\)
        - \(\text{Level}[w] = \text{Level}[u]+1\)
      - end if
  - end if
**Breadth First Search: BFS(v, G=(V,E))**

**Input:** Start node v, graph G (adjacency list) with n nodes V={1,...,n}

**Output:** Array Level[u] = distance from v to u

For each node u in V
- Explored[u]=0
- Level[u] = $\infty$ // No v→u path found yet

Q.Enqueue(v) // Queue Q (First In, First Out)

Level[v]=0

while(Q is not empty)
- u = Q.Dequeue()
  - if (Explored[u] = 0)
    - Explored[u]=1
  - foreach node w in u.AdjList
    - if (Level[w] = $\infty$)
      - Q.Enqueue(w)
      - Level[w] = Level[u]+1
end if

**Diagram:**
- Nodes: 1, 2, 3, 4, 5, 6
- Edges: 1→2, 1→3, 2→4, 3→5
- Initial State: Queue Q, Level 0, Explored 0

**Queue Q:**
- 1
- 2
- 3
- 4
- 5
- 6
Breadth First Search: \textbf{BFS}(v, G=(V,E))

**Input:** Start node \(v\), graph \(G\) (adjacency list) with \(n\) node \(V=\{1,\ldots,n\}\)

**Output:** Array \(\text{Level}[u]\) = distance from \(v\) to \(u\)

For each node \(u\) in \(V\)

\(\text{Explored}[u]=0\)

\(\text{Level}[u]=\infty\) \quad // No \(v\rightarrow u\) path found yet

\text{Q.Enqueue}(v) \quad // Queue \(Q\) (First In, First Out)

\(\text{Level}[v]=0\)

\textbf{while} (\(Q\) is not empty)

\(u = \text{Q.Dequeue}()\)

\textbf{if} (\text{Explored}[u] = 0)

\(\rightarrow \text{Explored}[u]=1\)

\(\rightarrow \textbf{foreach} \text{ node } w \text{ in } u.\text{AdjList}\)

\(\rightarrow \textbf{if} \ (\text{Level}[w] = \infty)\)

\(\rightarrow \text{Q.Enqueue}(w)\)

\(\rightarrow \text{Level}[w] = \text{Level}[u]+1\)

\textbf{end if}
Breadth First Search: \textbf{BFS}(v, G=(V,E))

\textbf{Input:} Start node \(v\), graph \(G\) (adjacency list) with \(n\) node \(V=\{1,\ldots,n\}\)

\textbf{Output:} Array \(\text{Level}[u] = \text{distance from } v \text{ to } u\)

\begin{itemize}
  \item For each node \(u\) in \(V\)
    \begin{itemize}
      \item \(\text{Explored}[u]=0\)
      \item \(\text{Level}[u] = \infty\) \hspace{1cm} // \text{No } v \rightarrow u \text{ path found yet}
    \end{itemize}
  \end{itemize}

\textbf{Q.Enqueue}(v) \hspace{1cm} // Queue \(Q\) (First In, First Out)

\begin{itemize}
  \item \(\text{Level}[v]=0\)
  \item while (\(Q\) is not empty)
    \begin{itemize}
      \item \(u = \text{Q.Dequeue}()\)
      \item if (\(\text{Explored}[u] = 0\))
        \begin{itemize}
          \item \(\text{Explored}[u]=1\)
        \end{itemize}
      \end{itemize}
      \(\rightarrow\) foreach node \(w\) in \(u\).AdjList
      \begin{itemize}
        \item if (\(\text{Level}[w] = \infty\))
          \begin{itemize}
            \item \(\text{Q.Enqueue}(w)\)
            \item \(\text{Level}[w] = \text{Level}[u]+1\)
          \end{itemize}
        \end{itemize}
    \end{itemize}
\end{itemize}
**Breadth First Search: BFS(v, G=(V,E))**

**Input:** Start node v, graph G (adjacency list) with n node V={1,...,n}

**Output:** Array Level[u] = distance from v to u

For each node u in V
- Explored[u]=0
- Level[u] = ∞ // No v→u path found yet

Q.Enqueue(v) // Queue Q (First In, First Out)

Level[v]=0

while(Q is not empty)
- u = Q.Dequeue()
- if (Explored[u] = 0)
  - Explored[u]=1

→ foreach node w in u.AdjList
  - if (Level[w] = ∞)
    → Q.Enqueue(w)
    → Level[w] = Level[u]+1

end if
Breadth First Search: \[ \text{BFS}(v, G=(V,E)) \]

**Input:** Start node \( v \), graph \( G \) (adjacency list) with \( n \) nodes \( V=\{1,\ldots,n\} \)

**Output:** Array \( \text{Level}[u] = \) distance from \( v \) to \( u \)

For each node \( u \) in \( V \)

- \( \text{Explored}[u]=0 \)
- \( \text{Level}[u] = \infty \) // No \( v\rightarrow u \) path found yet

\[ \text{Q.Enqueue}(v) \] // Queue \( Q \) (First In, First Out)

\( \text{Level}[v]=0 \)

\( \text{while}(Q \text{ is not empty}) \)

\( u = \text{Q.Dequeue}() \)

if \( (\text{Explored}[u] = 0) \)

- \( \text{Explored}[u]=1 \)
- foreach node \( w \) in \( u.\text{AdjList} \)

  if \( (\text{Level}[w] = \infty) \)

    - \( \text{Q.Enqueue}(w) \)
    - \( \text{Level}[w] = \text{Level}[u]+1 \)

end if
Breadth First Search: \textbf{BFS}(v, G=(V,E))

\textbf{Input}: Start node v, graph G (adjacency list) with n node \( V=\{1,\ldots,n\} \)

\textbf{Output}: Array \( \text{Level}[u] = \text{distance from } v \text{ to } u \)

For each node \( u \) in \( V \)
- \( \text{Explored}[u]=0 \)
- \( \text{Level}[u] = \infty \) // No \( v \rightarrow u \) path found yet
- \( \text{Q.Enqueue}(v) \) // Queue Q (First In, First Out)

\( \text{Level}[v]=0 \)

while (Q is not empty)
- \( u = \text{Q.Dequeue}() \)
- if (\( \text{Explored}[u] = 0 \))
  - \( \text{Explored}[u]=1 \)

→ foreach node \( w \) in \( u.\text{AdjList} \)
  - if (\( \text{Level}[w] = \infty \))
    - \( \text{Q.Enqueue}(w) \)
    - \( \text{Level}[w] = \text{Level}[u]+1 \)
  end if
**Breadth First Search: BFS(v, G=(V,E))**

**Input:** Start node v, graph G (adjacency list) with n node V={1,...,n}

**Output:** Array Level[u] = distance from v to u

For each node u in V

Explored[u]=0
Level[u] = \(\infty\) // No v→u path found yet
Q.Enqueue(v) // Queue Q (First In, First Out)
Level[v]=0

→ while(Q is not empty)

→ u = Q.Dequeue()

if (Explored[u] = 0)
    Explored[u]=1
    foreach node w in u.AdjList
        if (Level[w] = \(\infty\))
            Q.Enqueue(w)
            Level[w] = Level[u]+1
end if
Breadth First Search: \textbf{BFS}(v, G=(V,E))

\textbf{Input:} Start node v, graph G (adjacency list) with n node V={1,...,n}

\textbf{Output:} Array Level[u] = distance from v to u

For each node u in V
\begin{itemize}
    \item Explored[u]=0
    \item Level[u] = \infty \quad // No v\rightarrow u path found yet
\end{itemize}

Q.Enqueue(v) \quad // Queue Q (First In, First Out)

Level[v]=0

\begin{enumerate}
    \item \textbf{while (Q is not empty)}
    \item u = Q.Dequeue()
    \item if (Explored[u] = 0)
        \item Explored[u]=1
    \item \textbf{foreach} node w in u.AdjList
        \item if (Level[w] = \infty)
            \item Q.Enqueue(w)
            \item Level[w] = Level[u]+1
    \end{enumerate}

end if
Breadth First Search: \( \text{BFS}(v, G=(V,E)) \)

**Input:** Start node \( v \), graph \( G \) (adjacency list) with \( n \) node \( V=\{1,\ldots,n\} \)

**Output:** Array \( \text{Level}[u] = \text{distance from } v \text{ to } u \)

For each node \( u \) in \( V \)
- \( \text{Explored}[u]=0 \)
- \( \text{Level}[u] = \infty \)  // No \( v \rightarrow u \) path found yet
- \( \text{Q.Enqueue}(v) \)  // Queue \( Q \) (First In, First Out)
- \( \text{Level}[v]=0 \)

→ **while** (\( Q \) is not empty)
→ \( u = \text{Q.Dequeue}() \)
  - if (\( \text{Explored}[u] = 0 \))
    - \( \text{Explored}[u]=1 \)
    - foreach node \( w \) in \( u.\text{AdjList} \)
      - if (\( \text{Level}[w] = \infty \))
        - \( \text{Q.Enqueue}(w) \)
        - \( \text{Level}[w] = \text{Level}[u]+1 \)
  - end if
Breadth First Search: \textbf{BFS}(v, G=(V,E))

**Input:** Start node \( v \), graph \( G \) (adjacency list) with \( n \) node \( V=\{1,\ldots,n\} \)

**Output:** Array \( \text{Level}[u] = \text{distance from } v \text{ to } u \)

*For each* node \( u \) in \( V \)
  - \( \text{Explored}[u]=0 \)
  - \( \text{Level}[u] = \infty \) // No \( v \rightarrow u \) path found yet

\textbf{Q.Enqueue}(v) // Queue \( Q \) (First In, First Out)

- \( \text{Level}[v]=0 \)

\( \textbf{while}(Q \text{ is not empty}) \)

\( \textbf{u} = \textbf{Q.Dequeue}() \)

- if (\( \text{Explored}[u] = 0 \))
  - \( \text{Explored}[u]=1 \)

- \textbf{foreach} node \( w \) in \( u.\text{AdjList} \)
  - if (\( \text{Level}[w] = \infty \))
    - \( \textbf{Q.Enqueue}(w) \)
      - \( \text{Level}[w] = \text{Level}[u]+1 \)
  - end if
Breadth First Search: BFS(v, G=(V,E))

Input: Start node v, graph G (adjacency list) with n node V={1,...,n}

Output: Array Level[u] = distance from v to u

For each node u in V
  Explored[u]=0
  Level[u] = ∞  // No v→u path found yet
  Q.Enqueue(v)  // Queue Q (First In, First Out)
  Level[v]=0

while(Q is not empty)
  u = Q.Dequeue()
  if (Explored[u] = 0)
    Explored[u]=1
    foreach node w in u.AdjList
      if (Level[w] = ∞)
        Q.Enqueue(w)
        Level[w] = Level[u]+1
  end if
**Breadth First Search: BFS**(v, G=(V,E))

**Input:** Start node v, graph G (adjacency list) with n node V={1,...,n}

**Output:** Array Level[u] = distance from v to u

For each node u in V

- Explored[u]=0
- Level[u] = ∞ // No v→u path found yet

Q.Enqueue(v) // Queue Q (First In, First Out)

Level[v]=0

→ while(Q is not empty)

→ u = Q.Dequeue()

→ if (Explored[u] = 0)

→ Explored[u]=1

→ foreach node w in u.AdjList

→ if (Level[w] = ∞)

→ Q.Enqueue(w)

→ Level[w] = Level[u]+1

end if
Breadth First Search: $\text{BFS}(v, G=(V,E))$

**Input:** Start node $v$, graph $G$ (adjacency list) with $n$ nodes $V=\{1,\ldots,n\}$

**Output:** Array $\text{Level}[u] =$ distance from $v$ to $u$

For each node $u$ in $V$

- $\text{Explored}[u]=0$
- $\text{Level}[u] = \infty$ // No $v \rightarrow u$ path found yet

$\text{Q.Enqueue}(v)$ // Queue $Q$ (First In, First Out)

$\text{Level}[v]=0$

→ **while** ($Q$ is not empty)

→ $u = \text{Q.Dequeue}()$

if ($\text{Explored}[u] = 0$)

- $\text{Explored}[u]=1$

→ **foreach** node $w$ in $u.\text{AdjList}$

if ($\text{Level}[w] = \infty$)

- $\text{Q.Enqueue}(w)$

- $\text{Level}[w] = \text{Level}[u]+1$

end if

**Diagram:**

- Nodes and edges representing the graph $G$.
- Nodes labeled with their degrees or distances.
- $\text{Explored}$ and $\text{Level}$ annotations.

**Graph:**

- Nodes 1, 2, 3, 4, 5, 6 connected in a network.
- Distances indicated by numbers on the edges.

**Example:**

- $\text{Level}[1] = 0$
- $\text{Level}[2] = 1$
- $\text{Level}[3] = 2$
- $\text{Level}[4] = 3$
- $\text{Level}[5] = 2$
- $\text{Level}[6] = 4$
Breadth First Search: \( \text{BFS}(v, G=(V,E)) \)

**Input:** Start node \( v \), graph \( G \) (adjacency list) with \( n \) node \( V=\{1,\ldots,n\} \)

**Output:** Array \( \text{Level}[u] = \text{distance from } v \text{ to } u \)

1. For each node \( u \) in \( V \)
   - \( \text{Explored}[u]=0 \)
   - \( \text{Level}[u] = \infty \) // No \( v \to u \) path found yet
2. \( \text{Q.Enqueue}(v) \) // Queue \( Q \) (First In, First Out)
3. \( \text{Level}[v]=0 \)

4. while (\( Q \) is not empty)
5. \( u = \text{Q.Dequeue()} \)
6. if (\( \text{Explored}[u] = 0 \))
   - \( \text{Explored}[u]=1 \)
7. foreach node \( w \) in \( u.\text{AdjList} \)
   - if (\( \text{Level}[w] = \infty \))
     - \( \text{Q.Enqueue}(w) \)
     - \( \text{Level}[w] = \text{Level}[u]+1 \)
   - end if
**Breadth First Search: BFS(v, G=(V,E))**

**Input:** Start node v, graph G (adjacency list) with n node V={1,...,n}

**Output:** Array Level[u] = distance from v to u

For each node u in V
- Explored[u]=0
- Level[u] = ∞ // No v→u path found yet
- Q.Enqueue(v) // Queue Q (First In, First Out)
- Level[v]=0

→ while(Q is not empty)

→ u = Q.Dequeue()
  if (Explored[u] = 0)
    Explored[u]=1

→ foreach node w in u.AdjList
  if (Level[w] = ∞)
    Q.Enqueue(w)
    Level[w] = Level[u]+1

end if
**Input:** Start node \( v \), graph \( G \) (adjacency list) with \( n \) node \( V=\{1,\ldots,n\} \)

**Output:** Array \( \text{Level}[u] = \text{distance from } v \text{ to } u \)

- For each node \( u \) in \( V \)
  - \( \text{Explored}[u]=0 \)
  - \( \text{Level}[u] = \infty \) // No \( v \rightarrow u \) path found yet

- \( \text{Q.Enqueue}(v) \) // Queue \( Q \) (First In, First Out)

- \( \text{Level}[v]=0 \)

- **while** (\( Q \) is not empty)
  - \( u = \text{Q.Dequeue()} \)
  - if (\( \text{Explored}[u] = 0 \))
    - \( \text{Explored}[u]=1 \)
    - foreach node \( w \) in \( u.\text{AdjList} \)
      - if (\( \text{Level}[w] = \infty \))
        - \( \text{Q.Enqueue}(w) \)
        - \( \text{Level}[w] = \text{Level}[u]+1 \)
  - end if

**Diagram:**

- Graph \( G=(V,E) \) with nodes labeled from 1 to 6.
- Edges connecting nodes in a network.
- \( v \) marked as the starting vertex.
- Queue \( Q \) showing progression through the graph.
- **Explored** and **Level** attributes for each node.

**Done!**
Breadth First Search

BFS intuition. Explore outward from s in all possible directions, adding nodes one "layer" at a time.

BFS algorithm.
- $L_0 = \{ s \}$.
- $L_1 = \text{all neighbors of } L_0$.
- $L_2 = \text{all nodes that do not belong to } L_0 \text{ or } L_1, \text{ and that have an edge to a node in } L_1$.
- $L_{i+1} = \text{all nodes that do not belong to an earlier layer, and that have an edge to a node in } L_i$.

Theorem. For each i, $L_i$ consists of all nodes at distance exactly i from s. There is a path from s to t iff t appears in some layer.
**Property.** Let $T$ be a BFS tree of $G = (V, E)$, and let $(x, y)$ be an edge of $G$. Then the level of $x$ and $y$ differ by at most 1.
**Breadth First Search: Analysis**

**Theorem.** The above implementation of BFS runs in $O(m + n)$ time if the graph is given by its adjacency representation.

**Pf.**

- **Easy to prove $O(n^2)$ running time:**
  - at most $n$ lists $L[i]$
  - each node occurs on at most one list; for loop runs $\leq n$ times
  - when we consider node $u$, there are $\leq n$ incident edges $(u,v)$, and we spend $O(1)$ processing each edge

- **Actually runs in $O(m + n)$ time:**
  - when we consider node $u$, there are $\deg(u)$ incident edges $(u,v)$
  - total time processing edges is $\sum_{u \in V} \deg(u) = 2m$  

  

  each edge $(u, v)$ is counted exactly twice in sum: once in $\deg(u)$ and once in $\deg(v)$
**Connected Component**

**Connected component.** Find all nodes reachable from s.

![Graph with nodes and edges]

**Connected component containing node 1 = \{ 1, 2, 3, 4, 5, 6, 7, 8 \}.
Flood fill. Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.

- **Node:** pixel.
- **Edge:** two neighboring lime pixels.
- **Blob:** connected component of lime pixels.
Flood fill. Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.

- Node: pixel.
- Edge: two neighboring lime pixels.
- Blob: connected component of lime pixels.

![Diagram of Flood Fill process](image-url)
What problems can BFS solve?

- Check if an undirected graph is connected
- Determine *connected components* in an undirected graph
- Identify shortest path from source node to a destination node
- Is $G$ a tree? (Does $G$ have a cycle)
Finding connected components

Def. A connected component is a maximal set of connected vertices.

Sedgewick-Wayne 251 text
**Connected Component**

**Connected component.** Find all nodes reachable from $s$.

---

$R$ will consist of nodes to which $s$ has a path
Initially $R = \{s\}$
While there is an edge $(u, v)$ where $u \in R$ and $v \not\in R$
   Add $v$ to $R$
Endwhile

---

**Theorem.** Upon termination, $R$ is the connected component containing $s$.
- $BFS = \text{explore in order of distance from } s$.
- $DFS = \text{explore in a different way.}$
Suppose the shortest path from $u$ to $v$ has length $t > n/2$. Show that there exists some node $w \neq u, v$ s.t. deleting $w$ from $G$ disconnects $u$ and $v$.

**Hint:** Consider the BFS tree!

**Solution:** Run BFS($u$) to obtain levels $L_0, L_1, ...$

**Observation 1:** $v$ is in level $L_t$

**Observation 2:** $|L_0| + ... + |L_t| \leq n$

**Observation 3:** Must have some $0 < k < t$ with $|L_k| = 1$ by observation 2
Otherwise $|L_0| + |L_t| + (|L_1| + ... + |L_{t-1}|) \geq 2 + 2(t-1) > n$

**Observation 4:** Removing level $L_k$ disconnects $u$ and $v$
(BFS tree property $\rightarrow$ if $i < k < j$ then no edge connects $L_i$ and $L_j$)
Depth First Search (22.3+22.5; 251 text)

- DFS performs a recursive exploration of a graph
  - DFS follows a path until it is forced to back up – “backtracking”
- DFS operates on an adjacency list representation
- Most uses of DFS result in $O(n+m)$ time
- DFS be used on directed and undirected graphs

**Undirected graph**
- DFS partitions the edges into tree and back edges
- Assigns numbers to the vertices during exploration (e.g. DFS number, discovery number, finish number)