Def. An independent set of a graph \( G = (V, E) \) is a subset \( S \subseteq V \) of vertices such that for every \( u, v \in S \), we have \((u, v) \notin E\).
**Def.** An independent set of a graph \( G = (V, E) \) is a subset \( S \subseteq V \) of vertices such that for every \( u, v \in S \), we have \( (u, v) \notin E \).
Def. An independent set of a graph $G = (V, E)$ is a subset $S \subseteq V$ of vertices such that for every $u, v \in S$, we have $(u, v) \notin E$. 
### Maximum Independent Set Problem

**Input:** graph $G = (V, E)$

**Output:** the maximum independent set of $G$

```python
1: R;
2: for every set $S \subseteq V$
   3: b = true;
   4: for every $u, v \in S$
      5: if $(u, v) \in E$ then
         6: b = false
   7: if b and $|S| > |R|$ then $R \leftarrow S$
8: return $R$
```

Running time = $O(2^n n^2)$. 
Beyond Polynomial Time: $2^n$

### Maximum Independent Set Problem

**Input:** graph $G = (V, E)$

**Output:** the maximum independent set of $G$

**max-independent-set($G = (V, E)$)**

1. $R \leftarrow \emptyset$
2. **for** every set $S \subseteq V$ **do**
3. \> $b \leftarrow \text{true}$
4. \> **for** every $u, v \in S$ **do**
5. \> \> if $(u, v) \in E$ then $b \leftarrow \text{false}$
6. \> \> if $b$ and $|S| > |R|$ then $R \leftarrow S$
7. return $R$

Running time = $O(2^n n^2)$.
Beyond Polynomial Time: $n!$

**Hamiltonian Cycle Problem**

**Input:** a graph with $n$ vertices

**Output:** a cycle that visits each node exactly once, or say no such cycle exists
Beyond Polynomial Time: $n!$

**Hamiltonian Cycle Problem**

**Input:** a graph with $n$ vertices

**Output:** a cycle that visits each node exactly once, or say no such cycle exists
Beyond Polynomial Time: $n!$

**Hamiltonian**($G = (V, E)$)

1. for every permutation $(p_1, p_2, \cdots, p_n)$ of $V$ do
2. \hspace{1em} $b \leftarrow$ true
3. \hspace{1em} for $i \leftarrow 1$ to $n - 1$ do
4. \hspace{2em} if $(p_i, p_{i+1}) \notin E$ then $b \leftarrow$ false
5. \hspace{1em} if $(p_n, p_1) \notin E$ then $b \leftarrow$ false
6. \hspace{1em} if $b$ then return $(p_1, p_2, \cdots, p_n)$
7. return “No Hamiltonian Cycle”

Running time = $O(n! \times n)$
\(O(\log n)\) (Logarithmic) Running Time

**Binary search**

Input: sorted array \(A\) of size \(n\), an integer \(t\);
Output: whether \(t\) appears in \(A\).

E.g., search 35 in the following array:
$O(\log n)$ (Logarithmic) Running Time

- Binary search
  - Input: sorted array $A$ of size $n$, an integer $t$;
  - Output: whether $t$ appears in $A$. 

E.g., search 35 in the following array:
Binary search
- Input: sorted array $A$ of size $n$, an integer $t$;
- Output: whether $t$ appears in $A$.

E.g, search 35 in the following array:
**$O(\log n)$ (Logarithmic) Running Time**

- Binary search
  - Input: sorted array $A$ of size $n$, an integer $t$;
  - Output: whether $t$ appears in $A$.
- E.g, search 35 in the following array:

```
3  8  10  25  29  37  38  42  46  52  59  61  63  75  79
```
**$O(\log n)$ (Logarithmic) Running Time**

- Binary search
  - Input: sorted array $A$ of size $n$, an integer $t$;
  - Output: whether $t$ appears in $A$.
- E.g, search 35 in the following array:

```
  3  8  10  25  29  37  38  42  46  52  59  61  63  75  79
```
**$O(\log n)$ (Logarithmic) Running Time**

- **Binary search**
  - Input: sorted array $A$ of size $n$, an integer $t$;
  - Output: whether $t$ appears in $A$.
- E.g, search 35 in the following array:

```
3 8 10 25 29 37 38 42 46 52 59 61 63 75 79
```

42 > 35
Binary search
- Input: sorted array $A$ of size $n$, an integer $t$;
- Output: whether $t$ appears in $A$.

E.g, search 35 in the following array:
$O(\log n)$ (Logarithmic) Running Time

- Binary search
  - Input: sorted array $A$ of size $n$, an integer $t$;
  - Output: whether $t$ appears in $A$.
- E.g, search 35 in the following array:
Binary search

- Input: sorted array $A$ of size $n$, an integer $t$;
- Output: whether $t$ appears in $A$.

E.g., search 35 in the following array:
Binary search

Input: sorted array $A$ of size $n$, an integer $t$;
Output: whether $t$ appears in $A$.

E.g, search 35 in the following array:
$O(\log n)$ (Logarithmic) Running Time

- Binary search
  - Input: sorted array $A$ of size $n$, an integer $t$;
  - Output: whether $t$ appears in $A$.
- E.g, search 35 in the following array:

```
3  8  10  25  29  37  38  42  46  52  59  61  63  75  79
```
$O(\log n)$ (Logarithmic) Running Time

- **Binary search**
  - Input: sorted array $A$ of size $n$, an integer $t$;
  - Output: whether $t$ appears in $A$.
- E.g, search 35 in the following array:

```
3  8  10  25  29  37  38  42  46  52  59  61  63  75  79
```

$37 > 35$
Binary search

Input: sorted array \( A \) of size \( n \), an integer \( t \);
Output: whether \( t \) appears in \( A \).

E.g., search 35 in the following array:
$O(\log n)$ (Logarithmic) Running Time

Binary search

- Input: sorted array $A$ of size $n$, an integer $t$;
- Output: whether $t$ appears in $A$.

```plaintext
binary-search(A, n, t)
1:  $i \leftarrow 1, j \leftarrow n$
2:  while $i \leq j$ do
3:      $k \leftarrow \lfloor (i + j)/2 \rfloor$
4:      if $A[k] = t$ return true
5:      if $t < A[k]$ then $j \leftarrow k - 1$ else $i \leftarrow k + 1$
6:  return false
```
$O(\log n)$ (Logarithmic) Running Time

Binary search

- Input: sorted array $A$ of size $n$, an integer $t$;
- Output: whether $t$ appears in $A$.

```
binary-search(A, n, t)
1: $i \leftarrow 1, j \leftarrow n$
2: while $i \leq j$ do
3:     $k \leftarrow \lfloor (i + j)/2 \rfloor$
4:     if $A[k] = t$ return true
5:     if $t < A[k]$ then $j \leftarrow k - 1$ else $i \leftarrow k + 1$
6: return false
```

Running time = $O(\log n)$
Comparing the Orders

- Sort the functions from smallest to largest asymptotically:
  \( \log n, \ n \log n, \ n, \ n!, \ n^2, \ 2^n, \ e^n, \ n^n \)
- \( \log n = O(n) \)
Comparing the Orders

- Sort the functions from smallest to largest asymptotically:
  \( \log n, \ n \log n, \ n, \ n!, \ n^2, \ 2^n, \ e^n, \ n^n \)
- \( \log n = O(n) \)
- \( n = O(n^2) \)
Comparing the Orders

- Sort the functions from smallest to largest asymptotically:
  \( \log n, \ n \log n, \ n, \ n!, \ n^2, \ 2^n, \ e^n, \ n^n \)
- \( \log n = O(n) \)
- \( n = O(n \log n) \)
- \( n \log n = O(n^2) \)
Comparing the Orders

- Sort the functions from smallest to largest asymptotically:
  \( \log n, \ n \log n, \ n, \ n!, \ n^2, \ 2^n, \ e^n, \ n^n \)

- \( \log n = O(n) \)
- \( n = O(n \log n) \)
- \( n \log n = O(n^2) \)
- \( n^2 = O(n!) \)
Comparing the Orders

- Sort the functions from smallest to largest asymptotically:
  \( \log n, \ n \log n, \ n, \ n!, \ n^2, \ 2^n, \ e^n, \ n^n \)

- \( \log n = O(n) \)
- \( n = O(n \log n) \)
- \( n \log n = O(n^2) \)
- \( n^2 = O(2^n) \)
- \( 2^n = O(n!) \)
Comparing the Orders

- Sort the functions from smallest to largest asymptotically
  \( \log n, \ n \log n, \ n, \ n!, \ n^2, \ 2^n, \ e^n, \ n^n \)

- \( \log n = O(n) \)
- \( n = O(n \log n) \)
- \( n \log n = O(n^2) \)
- \( n^2 = O(2^n) \)
- \( 2^n = O(e^n) \)
- \( e^n = O(n!) \)
Comparing the Orders

- Sort the functions from smallest to largest asymptotically
  \( \log n, \ n \log n, \ n, \ n!, \ n^2, \ 2^n, \ e^n, \ n^n \)
- \( \log n = O(n) \)
- \( n = O(n \log n) \)
- \( n \log n = O(n^2) \)
- \( n^2 = O(2^n) \)
- \( 2^n = O(e^n) \)
- \( e^n = O(n!) \)
- \( n! = O(n^n) \)
Terminologies

When we talk about upper bound on running time:

- **Logarithmic time:** $O(\log n)$
- **Linear time:** $O(n)$
- **Quadratic time** $O(n^2)$
- **Cubic time** $O(n^3)$
- **Polynomial time:** $O(n^k)$ for some constant $k$
  - $O(n \log n) \subseteq O(n^{1.1})$. So, an $O(n \log n)$-time algorithm is also a polynomial time algorithm.
- **Exponential time:** $O(c^n)$ for some $c > 1$
- **Sub-linear time:** $o(n)$
- **Sub-quadratic time:** $o(n^2)$
<table>
<thead>
<tr>
<th>Goal of Algorithm Design</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Design algorithms to minimize the order of the running time.</strong></td>
</tr>
</tbody>
</table>

Using asymptotic analysis allows us to ignore the leading constants and lower order terms. This makes our life much easier! (E.g., the leading constant depends on the implementation, compiler and computer architecture of computer.)
Goal of Algorithm Design

- Design algorithms to minimize the order of the running time.

- Using asymptotic analysis allows us to ignore the leading constants and lower order terms.
Goal of Algorithm Design

- Design algorithms to minimize the order of the running time.
- Using asymptotic analysis allows us to ignore the leading constants and lower order terms.
- Makes our life much easier! (E.g., the leading constant depends on the implementation, compiler and computer architecture of computer.)
Q: Does ignoring the leading constant cause any issues?

- e.g, how can we compare an algorithm with running time $0.1n^2$ with an algorithm with running time $1000n$?
Q: Does ignoring the leading constant cause any issues?

- e.g., how can we compare an algorithm with running time $0.1n^2$ with an algorithm with running time $1000n$?

A:
Q: Does ignoring the leading constant cause any issues?

- e.g., how can we compare an algorithm with running time $0.1n^2$ with an algorithm with running time $1000n$?

A:

- Sometimes yes
Q: Does ignoring the leading constant cause any issues?

- e.g, how can we compare an algorithm with running time $0.1n^2$ with an algorithm with running time $1000n$?

A:
- Sometimes yes
- However, when $n$ is big enough, $1000n < 0.1n^2$
Q: Does ignoring the leading constant cause any issues?

- e.g., how can we compare an algorithm with running time $0.1n^2$ with an algorithm with running time $1000n$?

A:

- Sometimes yes
- However, when $n$ is big enough, $1000n < 0.1n^2$
- For “natural” algorithms, constants are not so big!
Q: Does ignoring the leading constant cause any issues?

- e.g., how can we compare an algorithm with running time $0.1n^2$ with an algorithm with running time $1000n$?

A:

- Sometimes yes
- However, when $n$ is big enough, $1000n < 0.1n^2$
- For “natural” algorithms, constants are not so big!
- So, for reasonably large $n$, algorithm with lower order running time beats algorithm with higher order running time.
Graph Basics

Lecturer: Kelin Luo

Department of Computer Science and Engineering
University at Buffalo
Outline

1. Graphs

2. Connectivity and Graph Traversal
   - Types of Graphs

3. Bipartite Graphs
   - Testing Bipartiteness

4. Topological Ordering
   - Applications: Word Ladder
Examples of Graphs

Figure: Road Networks

Figure: Social Networks

Figure: Internet

Figure: Transition Graphs
(Undirected) Graph $G = (V, E)$

- $V$: set of vertices (nodes);
- $E$: pairwise relationships among $V$;
- (undirected) graphs: relationship is symmetric, $E$ contains subsets of size 2
(Undirected) Graph $G = (V, E)$

- $V$: set of vertices (nodes);
  - $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$

- $E$: pairwise relationships among $V$;
  - (undirected) graphs: relationship is symmetric, $E$ contains subsets of size 2
  - $E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 4\}, \{2, 5\}, \{3, 5\}, \{3, 7\}, \{3, 8\}, \{4, 5\}, \{5, 6\}, \{7, 8\}\}$
Directed Graph $G = (V, E)$

- $V$: set of vertices (nodes);
  - $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$
- $E$: pairwise relationships among $V$;
  - directed graphs: relationship is asymmetric, $E$ contains ordered pairs
Directed Graph $G = (V, E)$

- $V$: set of vertices (nodes);
  - $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$

- $E$: pairwise relationships among $V$;
  - directed graphs: relationship is asymmetric, $E$ contains ordered pairs
  - $E = \{(1, 2), (1, 3), (3, 2), (4, 2), (2, 5), (5, 3), (3, 7), (3, 8), (4, 5), (5, 6), (6, 5), (8, 7)\}$
Abuse of Notations

- For (undirected) graphs, we often use \((i, j)\) to denote the set \(\{i, j\}\).
- We call \((i, j)\) an unordered pair; in this case \((i, j) = (j, i)\).

\[
E = \{(1, 2), (1, 3), (2, 3), (2, 4), (2, 5), (3, 5), (3, 7), (3, 8), (4, 5), (5, 6), (7, 8)\}
\]
- Social Network: Undirected
- Transition Graph: Directed
- Road Network: Directed or Undirected
- Internet: Directed or Undirected
Representation of Graphs

- **Adjacency matrix**
  - $n \times n$ matrix, $A[u, v] = 1$ if $(u, v) \in E$ and $A[u, v] = 0$ otherwise
  - $A$ is symmetric if graph is undirected

\[
\begin{array}{cccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
2 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
3 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\
4 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
5 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
7 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\
8 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
\end{array}
\]
Representation of Graphs

- **Adjacency matrix**
  - $n \times n$ matrix, $A[u, v] = 1$ if $(u, v) \in E$ and $A[u, v] = 0$ otherwise
  - $A$ is symmetric if graph is undirected

- **Linked lists**
  - For every vertex $v$, there is a linked list containing all neighbors of $v$. 
Representation of Graphs

- **Adjacency matrix**
  - $n \times n$ matrix, $A[u, v] = 1$ if $(u, v) \in E$ and $A[u, v] = 0$ otherwise
  - $A$ is symmetric if graph is undirected

- **Linked lists**
  - For every vertex $v$, there is a linked list containing all neighbors of $v$.
  - When graph is static, can use array of variant-length arrays.
Comparison of Two Representations

- Assuming we are dealing with undirected graphs
- $n$: number of vertices
- $m$: number of edges, assuming $n - 1 \leq m \leq n(n - 1)/2$
- $d_v$: number of neighbors of $v$

<table>
<thead>
<tr>
<th></th>
<th>Matrix</th>
<th>Linked Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory usage</td>
<td></td>
<td></td>
</tr>
<tr>
<td>time to check $(u, v) \in E$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>time to list all neighbors of $v$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Comparison of Two Representations

- Assuming we are dealing with undirected graphs
- \( n \): number of vertices
- \( m \): number of edges, assuming \( n - 1 \leq m \leq n(n - 1)/2 \)
- \( d_v \): number of neighbors of \( v \)

<table>
<thead>
<tr>
<th></th>
<th>Matrix</th>
<th>Linked Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory usage</td>
<td>( O(n^2) )</td>
<td></td>
</tr>
<tr>
<td>time to check ((u, v) \in E)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>time to list all neighbors of ( v )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Comparison of Two Representations

- Assuming we are dealing with undirected graphs
- $n$: number of vertices
- $m$: number of edges, assuming $n - 1 \leq m \leq n(n - 1)/2$
- $d_v$: number of neighbors of $v$

<table>
<thead>
<tr>
<th></th>
<th>Matrix</th>
<th>Linked Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory usage</td>
<td>$O(n^2)$</td>
<td>$O(m)$</td>
</tr>
<tr>
<td>time to check $(u, v) \in E$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>time to list all neighbors of $v$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Comparison of Two Representations

- Assuming we are dealing with undirected graphs
- \( n \): number of vertices
- \( m \): number of edges, assuming \( n - 1 \leq m \leq n(n - 1)/2 \)
- \( d_v \): number of neighbors of \( v \)

<table>
<thead>
<tr>
<th></th>
<th>Matrix</th>
<th>Linked Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory usage</td>
<td>( O(n^2) )</td>
<td>( O(m) )</td>
</tr>
<tr>
<td>time to check ((u, v) \in E)</td>
<td>( O(1) )</td>
<td></td>
</tr>
<tr>
<td>time to list all neighbors of ( v )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Comparison of Two Representations

- Assuming we are dealing with undirected graphs
- $n$: number of vertices
- $m$: number of edges, assuming $n - 1 \leq m \leq n(n - 1)/2$
- $d_v$: number of neighbors of $v$

<table>
<thead>
<tr>
<th></th>
<th>Matrix</th>
<th>Linked Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory usage</td>
<td>$O(n^2)$</td>
<td>$O(m)$</td>
</tr>
<tr>
<td>time to check $(u, v) \in E$</td>
<td>$O(1)$</td>
<td>$O(d_u)$</td>
</tr>
<tr>
<td>time to list all neighbors of $v$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Comparison of Two Representations

- Assuming we are dealing with undirected graphs
- \( n \): number of vertices
- \( m \): number of edges, assuming \( n - 1 \leq m \leq n(n - 1)/2 \)
- \( d_v \): number of neighbors of \( v \)

<table>
<thead>
<tr>
<th></th>
<th>Matrix</th>
<th>Linked Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory usage</td>
<td>( O(n^2) )</td>
<td>( O(m) )</td>
</tr>
<tr>
<td>time to check ((u, v) \in E)</td>
<td>( O(1) )</td>
<td>( O(d_u) )</td>
</tr>
<tr>
<td>time to list all neighbors of ( v )</td>
<td>( O(n) )</td>
<td></td>
</tr>
</tbody>
</table>
Comparison of Two Representations

- Assuming we are dealing with undirected graphs
- $n$: number of vertices
- $m$: number of edges, assuming $n - 1 \leq m \leq n(n - 1)/2$
- $d_v$: number of neighbors of $v$

<table>
<thead>
<tr>
<th></th>
<th>Matrix</th>
<th>Linked Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory usage</td>
<td>$O(n^2)$</td>
<td>$O(m)$</td>
</tr>
<tr>
<td>time to check $(u, v) \in E$</td>
<td>$O(1)$</td>
<td>$O(d_u)$</td>
</tr>
<tr>
<td>time to list all neighbors of $v$</td>
<td>$O(n)$</td>
<td>$O(d_v)$</td>
</tr>
</tbody>
</table>
Outline

1. Graphs

2. Connectivity and Graph Traversal
   - Types of Graphs

3. Bipartite Graphs
   - Testing Bipartiteness

4. Topological Ordering
   - Applications: Word Ladder
<table>
<thead>
<tr>
<th><strong>Connectivity Problem</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> graph $G = (V, E)$, (using linked lists)</td>
</tr>
<tr>
<td>two vertices $s, t \in V$</td>
</tr>
<tr>
<td><strong>Output:</strong> whether there is a path connecting $s$ to $t$ in $G$</td>
</tr>
</tbody>
</table>
Connectivity Problem

**Input:** graph $G = (V, E)$, (using linked lists)

two vertices $s, t \in V$

**Output:** whether there is a path connecting $s$ to $t$ in $G$

- Algorithm: starting from $s$, search for all vertices that are reachable from $s$ and check if the set contains $t$
Connectivity Problem

**Input:** graph \( G = (V, E) \), (using linked lists)

two vertices \( s, t \in V \)

**Output:** whether there is a path connecting \( s \) to \( t \) in \( G \)

- Algorithm: starting from \( s \), search for all vertices that are reachable from \( s \) and check if the set contains \( t \)
- Breadth-First Search (BFS)
Connectivity Problem

**Input:** graph $G = (V, E)$, (using linked lists)
  
two vertices $s, t \in V$

**Output:** whether there is a path connecting $s$ to $t$ in $G$

- Algorithm: starting from $s$, search for all vertices that are reachable from $s$ and check if the set contains $t$
  - Breadth-First Search (BFS)
  - Depth-First Search (DFS)
Breadth-First Search (BFS)

- Build layers $L_0, L_1, L_2, L_3, \cdots$
- $L_0 = \{s\}$
- $L_{j+1}$ contains all nodes that are not in $L_0 \cup L_1 \cup \cdots \cup L_j$ and have an edge to a vertex in $L_j$
Breadth-First Search (BFS)

- Build layers $L_0, L_1, L_2, L_3, \cdots$
- $L_0 = \{s\}$
- $L_{j+1}$ contains all nodes that are not in $L_0 \cup L_1 \cup \cdots \cup L_j$ and have an edge to a vertex in $L_j$
Breadth-First Search (BFS)

- Build layers $L_0, L_1, L_2, L_3, \cdots$
- $L_0 = \{s\}$
- $L_{j+1}$ contains all nodes that are not in $L_0 \cup L_1 \cup \cdots \cup L_j$ and have an edge to a vertex in $L_j$
Breadth-First Search (BFS)

- Build layers $L_0, L_1, L_2, L_3, \cdots$
- $L_0 = \{s\}$
- $L_{j+1}$ contains all nodes that are not in $L_0 \cup L_1 \cup \cdots \cup L_j$ and have an edge to a vertex in $L_j$
Breadth-First Search (BFS)

- Build layers $L_0, L_1, L_2, L_3, \ldots$
- $L_0 = \{s\}$
- $L_{j+1}$ contains all nodes that are not in $L_0 \cup L_1 \cup \cdots \cup L_j$ and have an edge to a vertex in $L_j$
Implementing BFS using a Queue

**BFS\( (s) \)**

1. \( \text{head} \leftarrow 1, \text{tail} \leftarrow 1, \text{queue}[1] \leftarrow s \)
2. mark \( s \) as “visited” and all other vertices as “unvisited”
3. \( \text{while head} \leq \text{tail} \) do
4. \( v \leftarrow \text{queue}[\text{head}], \text{head} \leftarrow \text{head} + 1 \)
5. \( \text{for all neighbors } u \text{ of } v \text{ do} \)
6. \( \quad \text{if } u \text{ is “unvisited” then} \)
7. \( \quad \text{tail} \leftarrow \text{tail} + 1, \text{queue}[\text{tail}] = u \)
8. \( \quad \text{mark } u \text{ as “visited”} \)

- Running time: \( O(n + m) \).
Example of BFS via Queue
Example of BFS via Queue
Example of BFS via Queue
Example of BFS via Queue
Example of BFS via Queue
Example of BFS via Queue
Example of BFS via Queue
Example of BFS via Queue

![Graph and queue diagram showing BFS traversal]

The diagram illustrates a breadth-first search (BFS) traversal of a graph. The graph consists of nodes labeled 1 to 8, and the queue shows the order in which nodes are visited. The traversal starts at node v and explores nodes at each level before moving to the next level.
Example of BFS via Queue
Example of BFS via Queue
Example of BFS via Queue
Example of BFS via Queue
Example of BFS via Queue
Example of BFS via Queue
Example of BFS via Queue
Example of BFS via Queue
Depth-First Search (DFS)

- Starting from \( s \)
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Depth-First Search (DFS)

- Starting from $s$
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Depth-First Search (DFS)

- Starting from \( s \)
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back

![Depth-First Search Diagram](image-url)
Depth-First Search (DFS)

- Starting from $s$
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Depth-First Search (DFS)

- Starting from $s$
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Depth-First Search (DFS)

- Starting from \( s \)
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Depth-First Search (DFS)

- Starting from $s$
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Depth-First Search (DFS)

- Starting from $s$
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex (“dead-end”), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Depth-First Search (DFS)

- Starting from $s$
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back

Diagram of a graph with vertices 1, 2, 3, 4, 5, 6, 7, 8, and edges connecting them.
Depth-First Search (DFS)

- Starting from \( s \)
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Depth-First Search (DFS)

- Starting from \( s \)
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex (“dead-end”), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Depth-First Search (DFS)

- Starting from \( s \)
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Implementing DFS using Recursion

DFS(s)
1: mark all vertices as “unvisited”
2: recursive-DFS(s)

recursive-DFS(v)
1: mark v as “visited”
2: for all neighbors u of v do
3: if u is unvisited then recursive-DFS(u)
Outline

1. Graphs

2. Connectivity and Graph Traversal
   - Types of Graphs

3. Bipartite Graphs
   - Testing Bipartiteness

4. Topological Ordering
   - Applications: Word Ladder
Path Graph (or Linear Graph)

**Def.** An undirected graph \( G = (V, E) \) is a path if the vertices can be listed in an order \( \{v_1, v_2, ..., v_n\} \) such that the edges are the \( \{v_i, v_{i+1}\} \) where \( i = 1, 2, ..., n - 1 \).

- Path graphs are connected graphs.
Def. An undirected graph $G = (V, E)$ is a cycle if its vertices can be listed in an order $v_1, v_2, \ldots, v_n$ such that the edges are the $\{v_i, v_{i+1}\}$ where $i = 1, 2, \ldots, n - 1$, plus the edge $\{v_n, v_1\}$.

- The degree of all vertices is 2.
Def. An undirected graph $G = (V, E)$ is a tree if any two vertices are connected by exactly one path. Or the graph is a connected acyclic graph.

- Most important type of special graphs: most computational problems are easier to solve on trees or lines.
**Def.** An undirected graph $G = (V, E)$ is a complete graph if each pair of vertices is joined by an edge.

- A complete graph contains all possible edges.
Def. An undirected graph $G = (V, E)$ is a planar graph if its vertices and edges can be drawn in a plane such that no two of the edges intersect.

Most computational problems have good solutions in a planar graph.
Directed Acyclic Graph (DAG)

Def. A directed graph $G = (V, E)$ is a directed acyclic graph if it is a directed graph with no directed cycles.

- DAG is equivalent to a partial ordering of nodes.
**Def.** An undirected graph $G = (V, E)$ is a **bipartite graph** if there is a partition of $V$ into two sets $L$ and $R$ such that for every edge $(u, v) \in E$, either $u \in L, v \in R$ or $v \in L, u \in R$. 