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: $o(n^2)$
Goal of Algorithm Design

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

\[
\begin{pmatrix}
1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\
1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
\end{pmatrix}
\]
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 $v$. 
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 \( n - 1 \leq m \leq \frac{n(n - 1)}{2} \)
- \( d_v \): 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
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 $G$
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 $L_0, L_1, L_2, L_3, \cdots$
- $L_0 = \{s\}$
- $L_{j+1}$ 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 $L_j$
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: while \( \text{head} \leq \text{tail} \) do
4: \( v \leftarrow \text{queue}[\text{head}], \text{head} \leftarrow \text{head} + 1 \)
5: for all neighbors \( u \) of \( v \) do
6: \hspace{2em} if \( u \) is “unvisited” then
7: \hspace{4.5em} \text{tail} \leftarrow \text{tail} + 1, \text{queue}[\text{tail}] = u
8: \hspace{2em} mark \( u \) as “visited”

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

Diagram of a graph showing nodes and edges, with a queue representation on the right side, illustrating the Breadth-First Search algorithm.
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 representation:

- Nodes: 1, 2, 3, 4, 5, 6, 7, 8
- Edges: (1, 2), (1, 3), (2, 3), (2, 4), (3, 4), (3, 5), (5, 6), (5, 8)

Queue representation:

- Queue: [1, 2, 3, 4, 5, 7]
- Head: 1
- Tail: 7

The vertex 'v' is the starting point for the breadth-first search.
Example of BFS via Queue
Example of BFS via Queue

1
2
3
4
5
6
7
8

v

tail
head
Example of BFS via Queue
Example of BFS via Queue
Example of BFS via Queue
Example of BFS via Queue

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
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 (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
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
**Def.** An undirected graph $G = (V, E)$ is a **path** if the 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 $i = 1, 2, \ldots, n - 1$.

- 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.
Complete Graph

**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.
Planar Graph

**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.
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 $v \in L, u \in R$. 
Outline

1. Graphs

2. Connectivity and Graph Traversal
   - Types of Graphs

3. Bipartite Graphs
   - Testing Bipartiteness

4. Topological Ordering
   - Applications: Word Ladder
Def. A 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 $v \in L, u \in R$. 
Testing Bipartiteness

- Taking an arbitrary vertex \( s \in V \)
Testing Bipartiteness

- Taking an arbitrary vertex \( s \in V \)
- Assuming \( s \in L \) w.l.o.g
Taking an arbitrary vertex $s \in V$

Assuming $s \in L$ w.l.o.g

Neighbors of $s$ must be in $R$
Testing Bipartiteness

- Taking an arbitrary vertex $s \in V$
- Assuming $s \in L$ w.l.o.g
- Neighbors of $s$ must be in $R$
- Neighbors of neighbors of $s$ must be in $L$

Report "not a bipartite graph" if contradiction was found

If $G$ contains multiple connected components, repeat above algorithm for each component
Testing Bipartiteness

- Taking an arbitrary vertex \( s \in V \)
- Assuming \( s \in L \) w.l.o.g
- Neighbors of \( s \) must be in \( R \)
- Neighbors of neighbors of \( s \) must be in \( L \)
- \( \ldots \)
Testing Bipartiteness

- Taking an arbitrary vertex \( s \in V \)
- Assuming \( s \in L \) w.l.o.g
- Neighbors of \( s \) must be in \( R \)
- Neighbors of neighbors of \( s \) must be in \( L \)
- \( \ldots \)
- Report “not a bipartite graph” if contradiction was found
Testing Bipartiteness

- Taking an arbitrary vertex \( s \in V \)
- Assuming \( s \in L \) w.l.o.g
- Neighbors of \( s \) must be in \( R \)
- Neighbors of neighbors of \( s \) must be in \( L \)
- \( \ldots \)
- Report “not a bipartite graph” if contradiction was found
- If \( G \) contains multiple connected components, repeat above algorithm for each component
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness

bad edges!