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
Testing Bipartiteness

- 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 $L$
- ...
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 the 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 \( L \)
- \( \cdots \)
- 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

bad edges!
Testing Bipartiteness using BFS

\[ \text{BFS}(s) \]

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: \( \text{v} \leftarrow \text{queue}[\text{head}], \text{head} \leftarrow \text{head} + 1 \)
5: for all neighbors \( u \) of \( v \) do
6: \hspace{1em} if \( u \) is “unvisited” then
7: \hspace{2em} \text{tail} \leftarrow \text{tail} + 1, \text{queue}[\text{tail}] = u
8: \hspace{2em} mark \( u \) as “visited”
test-bipartiteness(s)

1: head ← 1, tail ← 1, queue[1] ← s
2: mark s as “visited” and all other vertices as “unvisited”
3: color[s] ← 0
4: while head ≤ tail do
5:    v ← queue[head], head ← head + 1
6:    for all neighbors u of v do
7:       if u is “unvisited” then
8:           tail ← tail + 1, queue[tail] = u
9:           mark u as “visited”
10:      color[u] ← 1 − color[v]
11:    else if color[u] = color[v] then
12:       print(“G is not bipartite”) and exit
Testing Bipartiteness using BFS

1: mark all vertices as “unvisited”
2: for each vertex \( v \in V \) do
3: if \( v \) is “unvisited” then
4: test-bipartiteness\((v)\)
5: print(“\( G \) is bipartite”)
Testing Bipartiteness using BFS

1: mark all vertices as “unvisited”
2: for each vertex $v \in V$ do
3:    if $v$ is “unvisited” then
4:        test-bipartiteness($v$)
5:    print(“$G$ is bipartite”)

Obs. Running time of algorithm = $O(n + m)$
Testing Bipartiteness using DFS

**test-bipartiteness-DFS(\(s\))**

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

**recursive-test-DFS(\(v\))**

1: mark \(v\) as “visited”
2: **for** all neighbors \(u\) of \(v\) **do**
3: **if** \(u\) is unvisited **then**, recursive-test-DFS(\(u\))
Testing Bipartiteness using DFS

test-bipartiteness-DFS(s)

1: mark all vertices as “unvisited”
2: \text{color}[s] \leftarrow 0
3: recursive-test-DFS(s)

recursive-test-DFS(v)

1: mark v as “visited”
2: \textbf{for} all neighbors u of v \textbf{do}
3: \hspace{1em} \textbf{if} u is unvisited \textbf{then}
4: \hspace{2em} \text{color}[u] \leftarrow 1 - \text{color}[v], \text{recursive-test-DFS}(u)
5: \hspace{2em} \textbf{else if} \text{color}[u] = \text{color}[v] \textbf{then}
6: \hspace{3em} \text{print}(“G is not bipartite”) and exit
Testing Bipartiteness using DFS

1: mark all vertices as “unvisited”
2: for each vertex $v \in V$ do
3:    if $v$ is “unvisited” then
4:        test-bipartiteness-DFS($v$)
5: print(“$G$ is bipartite”)
Testing Bipartiteness using DFS

1: mark all vertices as “unvisited”
2: for each vertex $v \in V$ do
3:   if $v$ is “unvisited” then
4:     test-bipartiteness-DFS($v$)
5: print(“$G$ is bipartite”)

Obs. Running time of algorithm = $O(n + m)$
**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 $v \in L, u \in R$. 
**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 $v \in L, u \in R$.

**Obs.** Bipartite graph may contain cycles.
Bipartite Graph

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

**Obs.** Bipartite graph may contain cycles.

**Obs.** If a graph is a tree, then it is also a bipartite graph.
BFS and DFS

**Obs.** BFS and DFS naturally induce a tree.
Obs. BFS and DFS naturally induce a tree.

Obs. If $G$ is a tree, then BFS tree $=$ DFS tree.
BFS and DFS

**Obs.** BFS and DFS naturally induce a tree.

**Obs.** If $G$ is a tree, then BFS tree $=$ DFS tree.

**Obs.** If BFS tree $=$ DFS tree, then $G$ is a tree.
BFS and DFS

**Obs.** If BFS tree = DFS tree, then $G$ is a tree.

- True: simple, undirected graph
- Not True: directed graph
Outline

1. Graphs

2. Connectivity and Graph Traversal
   - Types of Graphs

3. Bipartite Graphs
   - Testing Bipartiteness

4. Topological Ordering
   - Applications: Word Ladder
Topological Ordering Problem

**Input:** a directed acyclic graph (DAG) $G = (V, E)$

**Output:** 1-to-1 function $\pi : V \rightarrow \{1, 2, 3 \cdots , n\}$, so that
- if $(u, v) \in E$ then $\pi(u) < \pi(v)$
Topological Ordering Problem

Input: a directed acyclic graph (DAG) \( G = (V, E) \)

Output: 1-to-1 function \( \pi : V \rightarrow \{1, 2, 3 \cdots, n\} \), so that
- if \((u, v) \in E\) then \(\pi(u) < \pi(v)\)
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- **Algorithm**: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.

Q: How to make the algorithm as efficient as possible?
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.

Q: How to make the algorithm as efficient as possible?

A:
- Use linked-lists of outgoing edges
- Maintain the in-degree $d_v$ of vertices
- Maintain a queue (or stack) of vertices $v$ with $d_v = 0$
topological-sort($G$)

1. let $d_v \leftarrow 0$ for every $v \in V$
2. for every $v \in V$ do
3.   for every $u$ such that $(v, u) \in E$ do
4.     $d_u \leftarrow d_u + 1$
5. $S \leftarrow \{v : d_v = 0\}$, $i \leftarrow 0$
6. while $S \neq \emptyset$ do
7.    $v \leftarrow$ arbitrary vertex in $S$, $S \leftarrow S \setminus \{v\}$
8.    $i \leftarrow i + 1$, $\pi(v) \leftarrow i$
9. for every $u$ such that $(v, u) \in E$ do
10.   $d_u \leftarrow d_u - 1$
11. if $d_u = 0$ then add $u$ to $S$
12. if $i < n$ then output “not a DAG”

- $S$ can be represented using a queue or a stack
- Running time $= O(n + m)$
### $S$ as a Queue or a Stack

<table>
<thead>
<tr>
<th>DS</th>
<th>Queue</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>$head \leftarrow 1$, $tail \leftarrow 0$</td>
<td>$top \leftarrow 0$</td>
</tr>
<tr>
<td>Non-Empty?</td>
<td>$head \leq tail$</td>
<td>$top &gt; 0$</td>
</tr>
<tr>
<td>Add($v$)</td>
<td>$tail \leftarrow tail + 1$</td>
<td>$top \leftarrow top + 1$</td>
</tr>
<tr>
<td></td>
<td>$S[tail] \leftarrow v$</td>
<td>$S[top] \leftarrow v$</td>
</tr>
<tr>
<td>Retrieve $v$</td>
<td>$v \leftarrow S[head]$</td>
<td>$v \leftarrow S[top]$</td>
</tr>
<tr>
<td></td>
<td>$head \leftarrow head + 1$</td>
<td>$top \leftarrow top - 1$</td>
</tr>
</tbody>
</table>
Example

Queue:
a b c d e f g

Degree:
<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>degree</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>
Example

![Diagram of a graph with nodes labeled a, b, c, d, e, f, g.]

A queue labeled a, b, c, d, e, f, g with arrows indicating direction.

Degree distribution:

<table>
<thead>
<tr>
<th>degree</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>
Example

- **Queue:**
  - **Tail:** e
  - **Head:** a
  - **Elements:** a, b, c, d, e, f, g

- **Degree:**
  - a: 0, b: 0, c: 0, d: 1, e: 2, f: 1, g: 3
Example

Queue:

\[
\begin{array}{ccccccc}
\text{tail} & \text{head} \\
\hline
\text{queue:} & a & b & c & d & e & f & g \\
\text{degree} & 0 & 0 & 0 & 1 & 2 & 1 & 3 \\
\end{array}
\]
Example

Queue:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Degree:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>
Example

- **c**
- **d**
- **f**
- **e**
- **g**

**Queue:**

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>degree</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

**Head**

**Tail**
Example

Queue:

```
queue:  a  b  c
```

degree:

```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>
```
Example

- **Queue:**
  - Queue: $a \ b \ c$
  - Tail
  - Head

- **Degree:**
  - Degree: $0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 3$
Example

queue:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
</tr>
</tbody>
</table>

degree

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>
Example

queue:

\[
\begin{array}{cccccc}
\text{head} & & & & & \text{tail} \\
\text{queue:} & a & b & c & d & f \\
\end{array}
\]

\[
\begin{array}{cccccccc}
\text{degree} & a & b & c & d & e & f & g \\
0 & 0 & 0 & 0 & 1 & 0 & 3 \\
\end{array}
\]

\[
\begin{array}{ccc}
\text{graph:} & e & \rightarrow d & \rightarrow f & \rightarrow g \\
\end{array}
\]
Example

queue: \[ a \quad b \quad c \quad d \quad f \]

\[
\begin{array}{c|cccccccc}
\text{degree} & a & b & c & d & e & f & g \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \\
\end{array}
\]
Example

queue: \[ a \ b \ c \ d \ f \ e \]

<table>
<thead>
<tr>
<th>degree</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>
Example

queue: \[ \begin{array}{ccccccc} a & b & c & d & f & e \end{array} \]

degree

\[
\begin{array}{cccccccc}
\text{degree} & a & b & c & d & e & f & g \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 2
\end{array}
\]
Example

queue: a b c d f e

degree | a | b | c | d | e | f | g
|------|---|---|---|---|---|---|---|
|      | 0 | 0 | 0 | 0 | 0 | 0 | 1
Example

queue:

\[
\begin{array}{ccccccc}
    & a & b & c & d & f & e \\
\end{array}
\]

degree:

\[
\begin{array}{cccccccc}
    & a & b & c & d & e & f & g \\
    \text{degree} & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\end{array}
\]
Example

queue: $a \ b \ c \ d \ f \ e$

<table>
<thead>
<tr>
<th>degree</th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>$d$</th>
<th>$e$</th>
<th>$f$</th>
<th>$g$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Example

Queue:

\[
\begin{array}{cccccccc}
\text{a} & \text{b} & \text{c} & \text{d} & \text{f} & \text{e} & \text{g} \\
\text{g} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]
Example

Queue: $a \ b \ c \ d \ f \ e \ g$

Degree: $0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0$
Outline

1. Graphs

2. Connectivity and Graph Traversal
   - Types of Graphs

3. Bipartite Graphs
   - Testing Bipartiteness

4. Topological Ordering
   - Applications: Word Ladder
Def. Word: A string formed by letters.

Def. Adjacency words: Word $A$ and $B$ are adjacent if they differ in exactly one letter.

E.g. word and work; tell and tall; askbe and askee.
Def. Word Ladder: Players start with one word, and in a series of steps, change or transform that word into another word.
**Def.** Word Ladder: Players start with one word, and in a series of steps, change or transform that word into another word.

- The objective is to make the change in the smallest number of steps, with each step involving changing a **single letter** of the word to create a new valid word.
**Word Ladder Problem**

**Input:** Two words $S$ and $T$, a list of words $A = \{W_1, W_2, \ldots, W_k\}$.

**Output:** “The smallest word ladder” if we can change $S$ to $T$ by moving between adjacency words in $A \cup \{S, T\}$; Otherwise, “No word ladder”.

Example:

- $S=\text{"a e f g h"}$, $T = \text{"d l m i h"}$
- $W_1=\text{"a e f i h"}$, $W_2 = \text{"a e m g h"}$, $W_3=\text{"d l f i h"}$
- $W_4 = \text{"s e f i h"}$, $W_5=\text{"a d f g h"}$, $W_6 = \text{"d e m i h"}$
- $W_7=\text{"d e f i h"}$, $W_8 = \text{"d e m g h"}$, $W_9 = \text{"s e m i h"}$
Example:
- $S = \text{“a e f g h”}$, $T = \text{“d l m i h”}$
- $W_1 = \text{“a e f i h”}$, $W_2 = \text{“a e m g h”}$, $W_3 = \text{“d l f i h”}$
- $W_4 = \text{“s e f i h”}$, $W_5 = \text{“a d f g h”}$, $W_6 = \text{“d e m i h”}$
- $W_7 = \text{“d e f i h”}$, $W_8 = \text{“d e m g h”}$, $W_9 = \text{“s e m i h”}$

- Each vertex corresponds to a word.
- Two vertices are adjacent if the corresponding words are adjacent.
Each vertex corresponds to a word.

Two vertices are adjacent if the corresponding words are adjacent.

Hints: Given vertex $v$, check its nearest neighbor.