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 \( R \)

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 $L$
- \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 $L$
- ...$
- 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 \( L \)
- \( \cdots \)
- 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
Test Bipartiteness

bad edges!
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: \( \text{while} \ \text{head} \leq \text{tail} \ \text{do} \)
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: \( \text{if} \ u \ \text{is “unvisited” then} \)
7: \( \text{tail} \leftarrow \text{tail} + 1, \text{queue}[\text{tail}] = u \)
8: \( \text{mark} \ u \ \text{as “visited”} \)
Testing Bipartiteness using BFS

test-bipartiteness($s$)

1. $head \leftarrow 1$, $tail \leftarrow 1$, $queue[1] \leftarrow s$
2. mark $s$ as “visited” and all other vertices as “unvisited”
3. $color[s] \leftarrow 0$
4. while $head \leq tail$ do
5. \hspace{1em} $v \leftarrow queue[head]$, $head \leftarrow head + 1$
6. \hspace{1em} for all neighbors $u$ of $v$ do
7. \hspace{2em} if $u$ is “unvisited” then
8. \hspace{3em} $tail \leftarrow tail + 1$, $queue[tail] = u$
9. \hspace{3em} mark $u$ as “visited”
10. \hspace{2em} $color[u] \leftarrow 1 - color[v]$
11. \hspace{1em} else if $color[u] = color[v]$ then
12. \hspace{1.5em} 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: \hspace{1em} if $v$ is “unvisited” then
4: \hspace{2em} 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: color[s] ← 0
3: 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
4:     color[u] ← 1 − color[v], recursive-test-DFS(u)
5:   else if color[u] = color[v] then
6:     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 $v \in L, u \in R$. 
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.
**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.
**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.
1. Graphs

2. Connectivity and Graph Traversal
   - Types of Graphs

3. Bipartite Graphs
   - Testing Bipartiteness

4. Topological Ordering
Topological Ordering Problem

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

**Output:** 1-to-1 function $\pi : V \to \{1, 2, 3 \cdots, n\}$, 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 $\pi : V \to \{1, 2, 3 \cdots, n\}$, 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.
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.

![Diagram](attachment:image.png)
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.
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.
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.

![Graph diagram](image-url)
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.
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?
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 $= O(n + m)$