Today

- Graphs
  - Topological Sort
  - Graph Traversals
Topological Sort

Problem: Given a DAG $G = (V, E)$, output all the vertices in order such that no vertex appears before any other vertex that has an edge to it.

Example input:

Example output:

142, 126, 143, 311, 331, 332, 312, 341, 351, 333, 440, 352
Valid Topological Sorts:
Valid Topological Sorts:

0
Valid Topological Sorts:

0, 1
Valid Topological Sorts:
0, 1, 3
Valid Topological Sorts:

0, 1, 3, 2
Valid Topological Sorts:

0, 1, 3, 2, 4
Valid Topological Sorts:

0, 1, 3, 2, 4
0, 3, 1, 2, 4 ??
Valid Topological Sorts:

0, 1, 3, 2, 4

0, 3, 1, 2, 4 – 3 appears before 1!!!
Valid Topological Sorts:

- 0, 1, 3, 2, 4
- 0, 1, 2, 3, 4
- 1, 0, 2, 3, 4
- 1, 0, 3, 2, 4
- 1, 2, 0, 3, 4
Questions and comments

• Why do we perform topological sorts only on DAGs?

• Is there always a unique answer?

• What DAGs have exactly 1 answer?

• Terminology: A DAG represents a partial order and a topological sort produces a total order that is consistent with it
Questions and comments

• Why do we perform topological sorts only on DAGs?
  – Directed – direction shows relationship/dependency

• Is there always a unique answer?

• What DAGs have exactly 1 answer?

• Terminology: A DAG represents a partial order and a topological sort produces a total order that is consistent with it
Questions and comments

- Why do we perform topological sorts only on DAGs?
  - Directed – direction shows relationship/dependency
  - Acyclic – cycle means no ordering is possible
- Is there always a unique answer?

- What DAGs have exactly 1 answer?

- Terminology: A DAG represents a partial order and a topological sort produces a total order that is consistent with it
Questions and comments

• Why do we perform topological sorts only on DAGs?
  – Directed – direction shows relationship/dependency
  – Acyclic – cycle means no ordering is possible

• Is there always a unique answer?
  – No, there can be 1 or more answers; depends on the graph

• What DAGs have exactly 1 answer?
  – Lists

• Terminology: A DAG represents a partial order and a topological sort produces a total order that is consistent with it
Topological Sort Uses

- Figuring out how to finish your degree
- Figuring the order in which to implement classes for Project 2
- Determining the order to compile files using a Makefile
- In general, taking a dependency graph and coming up with an order of execution
A First Algorithm for Topological Sort

1. Label (“mark”) each vertex with its in-degree
   - Think “write in a field in the vertex”
   - Could also do this via a data structure (e.g., array) on the side

2. While there are vertices not yet output:
   a) Choose a vertex $v$ with labeled with in-degree of 0
   b) Output $v$ and conceptually remove it from the graph
   c) For each vertex $w$ adjacent to $v$ (i.e. $w$ such that $(v,w)$ in $E$), decrement the in-degree of $w$
Valid Topological Sorts:

- 0, 1, 3, 2, 4
- 0, 1, 2, 3, 4
- 1, 0, 2, 3, 4
- 1, 0, 3, 2, 4
- 1, 2, 0, 3, 4
A First Algorithm for Topological Sort

1. Label (“mark”) each vertex with its in-degree
   - Think “write in a field in the vertex”
   - Could also do this via a data structure (e.g., array) on the side

2. While there are vertices not yet output:
   a) Choose a vertex \( v \) with labeled with in-degree of 0
   b) Output \( v \) and *conceptually* remove it from the graph
   c) For each vertex \( w \) adjacent to \( v \) (i.e. \( w \) such that \((v,w)\) in \( E \)), decrement the in-degree of \( w \)
A First Algorithm for Topological Sort

1. Label (“mark”) each vertex with its in-degree
   - Think “write in a field in the vertex”
   - Could also do this via a data structure (e.g., array) on the side

2. While there are vertices not yet output:
   a) Choose a vertex \( v \) with labeled with in-degree of 0
   b) Output \( v \) and conceptually remove it from the graph
   c) For each vertex \( w \) adjacent to \( v \) (i.e. \( w \) such that \((v,w)\) in \( E \)),
      decrement the in-degree of \( w \)

\[
\begin{array}{c|c|c}
\text{In-degree} & 0 & 1 \\
\hline
0 & 0 & 1 \\
1 & 2 & 1 \\
2 & 2 & 2 \\
3 & 2 & \_ \\
4 & 2 & \_ \\
\end{array}
\]
A First Algorithm for Topological Sort

1. Label ("mark") each vertex with its in-degree
   - Think “write in a field in the vertex”
   - Could also do this via a data structure (e.g., array) on the side

2. While there are vertices not yet output:
   a) Choose a vertex $v$ with labeled with in-degree of 0
   b) Output $v$ and conceptually remove it from the graph
   c) For each vertex $w$ adjacent to $v$ (i.e. $w$ such that $(v,w)$ in $E$),
      decrement the in-degree of $w$
A First Algorithm for Topological Sort

1. Label (“mark”) each vertex with its in-degree
   - Think “write in a field in the vertex”
   - Could also do this via a data structure (e.g., array) on the side
2. While there are vertices not yet output:
   a) Choose a vertex \( v \) with labeled with in-degree of 0
   b) Output \( v \) and conceptually remove it from the graph
   c) For each vertex \( w \) adjacent to \( v \) (i.e. \( w \) such that \((v, w)\) in \( E \)), decrement the in-degree of \( w \)

<table>
<thead>
<tr>
<th>In-degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

Output: 0
A First Algorithm for Topological Sort

1. Label (“mark”) each vertex with its in-degree
   - Think “write in a field in the vertex”
   - Could also do this via a data structure (e.g., array) on the side

2. While there are vertices not yet output:
   a) Choose a vertex \( v \) with labeled with in-degree of 0
   b) Output \( v \) and conceptually remove it from the graph
   c) For each vertex \( w \) adjacent to \( v \) (i.e. \( w \) such that \((v,w) \in E\)), decrement the in-degree of \( w \)

<table>
<thead>
<tr>
<th>In-degree</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>3</td>
<td>/</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>2</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td>/</td>
</tr>
</tbody>
</table>

Output: 0
A First Algorithm for Topological Sort

1. Label (“mark”) each vertex with its in-degree
   - Think “write in a field in the vertex”
   - Could also do this via a data structure (e.g., array) on the side

2. While there are vertices not yet output:
   a) Choose a vertex $v$ with labeled with in-degree of 0
   b) Output $v$ and conceptually remove it from the graph
   c) For each vertex $w$ adjacent to $v$ (i.e. $w$ such that $(v,w)$ in $E$), decrement the in-degree of $w$
A First Algorithm for Topological Sort

1. Label ("mark") each vertex with its in-degree
   - Think "write in a field in the vertex"
   - Could also do this via a data structure (e.g., array) on the side

2. While there are vertices not yet output:
   a) Choose a vertex \( v \) with labeled with in-degree of 0
   b) Output \( v \) and conceptually remove it from the graph
   c) For each vertex \( w \) adjacent to \( v \) (i.e. \( w \) such that \((v,w)\) in \( E \)),
      decrement the in-degree of \( w \)

   ![Diagram](image)

   **In-degree**
   
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

   Output: 0, 1
A First Algorithm for Topological Sort

1. Label ("mark") each vertex with its in-degree
   - Think “write in a field in the vertex”
   - Could also do this via a data structure (e.g., array) on the side

2. While there are vertices not yet output:
   a) Choose a vertex \( v \) with labeled with in-degree of 0
   b) Output \( v \) and conceptually remove it from the graph
   c) For each vertex \( w \) adjacent to \( v \) (i.e. \( w \) such that \((v,w)\) in \( E\)), decrement the in-degree of \( w \)

In-degree

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3 /</td>
<td>2</td>
<td>4 /</td>
<td>4 /</td>
<td>/</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3 /</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>2</td>
<td>4 /</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>3</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>4</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
</tbody>
</table>

Output: 0, 1
A First Algorithm for Topological Sort

1. Label (“mark”) each vertex with its in-degree
   - Think “write in a field in the vertex”
   - Could also do this via a data structure (e.g., array) on the side

2. While there are vertices not yet output:
   a) Choose a vertex $v$ with labeled with in-degree of 0
   b) Output $v$ and conceptually remove it from the graph
   c) For each vertex $w$ adjacent to $v$ (i.e. $w$ such that $(v,w)$ in $E$), decrement the in-degree of $w$

In-degree

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

Output: 0, 1
A First Algorithm for Topological Sort

1. Label ("mark") each vertex with its in-degree
   - Think "write in a field in the vertex"
   - Could also do this via a data structure (e.g., array) on the side

2. While there are vertices not yet output:
   a) Choose a vertex \( v \) with labeled with in-degree of 0
   b) Output \( v \) and conceptually remove it from the graph
   c) For each vertex \( w \) adjacent to \( v \) (i.e. \( w \) such that \( (v,w) \) in \( E \)), decrement the in-degree of \( w \)

Output: 0, 1, 2
A First Algorithm for Topological Sort

1. Label (“mark”) each vertex with its in-degree
   - Think “write in a field in the vertex”
   - Could also do this via a data structure (e.g., array) on the side

2. While there are vertices not yet output:
   a) Choose a vertex $v$ with labeled with in-degree of 0
   b) Output $v$ and conceptually remove it from the graph
   c) For each vertex $w$ adjacent to $v$ (i.e. $w$ such that $(v,w)$ in $E$), decrement the in-degree of $w$

```
In-degree

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>3/</td>
<td>2</td>
<td>4/</td>
<td>4/</td>
<td>/</td>
</tr>
<tr>
<td>2</td>
<td>3/</td>
<td>3/</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>/</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>/</td>
</tr>
</tbody>
</table>
```

Output: 0, 1, 2
A First Algorithm for Topological Sort

1. Label (“mark”) each vertex with its in-degree
   - Think “write in a field in the vertex”
   - Could also do this via a data structure (e.g., array) on the side

2. While there are vertices not yet output:
   a) Choose a vertex \( v \) with labeled with in-degree of 0
   b) Output \( v \) and conceptually remove it from the graph
   c) For each vertex \( w \) adjacent to \( v \) (i.e. \( w \) such that \( (v, w) \) in \( E \)), decrement the in-degree of \( w \)

\[
\begin{array}{c|c}
\text{In-degree} & \text{Vertex} \\
0 & 0 \\
1 & 0 \\
2 & 0 \\
3 & 0 \\
4 & 1 \\
\end{array}
\]

Output: 0, 1, 2
A First Algorithm for Topological Sort

1. Label ("mark") each vertex with its in-degree
   - Think "write in a field in the vertex"
   - Could also do this via a data structure (e.g., array) on the side

2. While there are vertices not yet output:
   a) Choose a vertex \( v \) with labeled with in-degree of 0
   b) Output \( v \) and conceptually remove it from the graph
   c) For each vertex \( w \) adjacent to \( v \) (i.e. \( w \) such that \((v,w) \in E\)),
      decrement the in-degree of \( w \)

Output: 0, 1, 2, 3

<table>
<thead>
<tr>
<th>In-degree</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

2/23/2018
A First Algorithm for Topological Sort

1. Label ("mark") each vertex with its in-degree
   - Think "write in a field in the vertex"
   - Could also do this via a data structure (e.g., array) on the side

2. While there are vertices not yet output:
   a) Choose a vertex $v$ with labeled with in-degree of 0
   b) Output $v$ and conceptually remove it from the graph
   c) For each vertex $w$ adjacent to $v$ (i.e. $w$ such that $(v,w)$ in $E$), decrement the in-degree of $w$

Output: 0, 1, 2, 3
A First Algorithm for Topological Sort

1. Label (“mark”) each vertex with its in-degree
   - Think “write in a field in the vertex”
   - Could also do this via a data structure (e.g., array) on the side
2. While there are vertices not yet output:
   a) Choose a vertex $v$ with labeled with in-degree of 0
   b) Output $v$ and conceptually remove it from the graph
   c) For each vertex $w$ adjacent to $v$ (i.e. $w$ such that $(v,w)$ in $E$), decrement the in-degree of $w$

Output: 0, 1, 2, 3, 4
Example

Node: 126 142 143 311 312 331 332 333 341 351 352 440
Removed?
In-degree: 0 0 2 1 2 1 1 2 1 1 1 1
Example

Node: 126 142 143 311 312 331 332 333 341 351 352 440
Removed? x
In-degree: 0 0 2 1 2 1 1 2 1 1 1 1

Output: 126
Example

Node: 126 142 143 311 312 331 332 333 341 351 352 440
Removed?  x  x
In-degree: 0 0 2 1 2 1 1 2 1 1 1 1
           1
           0

Output: 126
        142
Example

Node: 126 142 143 311 312 331 332 333 341 351 352 440

Removed?  x  x  x

In-degree:  0  0  2  1  2  1  1  2  1  1  1  1

Output: 126
         142
         143

2/23/2018
Example

Node: 126 142 143 311 312 331 332 333 341 351 352 440
Removed?  x  x  x  x  x
In-degree: 0 0 2 1 2 1 1 2 1 1 1 1

Output: 126 142 143 311
**Example**

Node:  126 142 143 311 312 331 332 333 341 351 352 440

Removed?  x  x  x  x  x  x

In-degree:  0  0  2  1  2  1  1  2  1  1  1  1

Output:  126 142 143 311 331
Example

Node: 126 142 143 311 312 331 332 333 341 351 352 440
Removed?  x  x  x  x  x  x  x  x
In-degree: 0 0 2 1 2 1 1 2 1 1 1 1
          1 0 1 0 0 1 0 0 0 0 0
                  0  0

Output: 126
        142
        143
        311
        331
        332
Example

Node: 126 142 143 311 312 331 332 333 341 351 352 440
Removed? x x x x x x x x x
In-degree: 0 0 2 1 2 1 1 2 1 1 1 1

Output: 126 142 143 311 331 332 312
Example

Output: 126
142
143
311
331
332
312
341

Node: 126 142 143 311 312 331 332 333 341 351 352 440
Removed? x x x x x x x x x x x
In-degree: 0 0 2 1 2 1 1 2 1 1 1 1
           1 0 1 0 0 1 0 0 0 0 0 0
           0 0
Example

Node: 126 142 143 311 312 331 332 333 341 351 352 440
Removed? x x x x x x x x x x x
In-degree: 0 0 2 1 2 1 1 2 1 1 1 1
           1 0 1 0 0 1 0 0 0 0 0 0
           0 0 0 0 0

Output: 126
         142
         143
         311
         331
         332
         312
         341
         351
         352
         440

2/23/2018
Example

Node: 126 142 143 311 312 331 332 333 341 351 352 440
Removed? x x x x x x x x x x x x x x
In-degree: 0 0 2 1 2 1 1 2 1 1 1 1

Output: 126 142 143 311 331 332 333 341 351 352 440

2/23/2018
A couple of things to note

• Needed a vertex with in-degree of 0 to start
  – No cycles
• Ties between vertices with in-degrees of 0 can be broken arbitrarily
  – Potentially many different correct orders
**Topological Sort: Running time?**

```java
labelEachVertexWithItsInDegree();
for(ctr=0; ctr < numVertices; ctr++){
    v = findNewVertexOfDegreeZero();
    put v next in output
    for each w adjacent to v
        w.indegree--;
}
```

You can use a helper variable `d` – the out degree of a vertex
Topological Sort: Running time?

labelEachVertexWithItsInDegree(); \( O(V+E) \)
for(ctr=0; ctr < numVertices; ctr++){
    v = findNewVertexOfDegreeZero();
    put v next in output
    for each w adjacent to v
        w.indegree--;
}

You can use a helper variable \( d \) – the out degree of a vertex
Topological Sort: Running time?

```
labelEachVertexWithItsInDegree(); O(V+E)
for(ctr=0; ctr < numVertices; ctr++){
  V times
  v = findNewVertexOfDegreeZero();
  put v next in output
  for each w adjacent to v
    w.indegree--;
}
```

You can use a helper variable \(d\) – the out degree of a vertex
Topological Sort: Running time?

```java
labelEachVertexWithItsInDegree(); O(V+E)
for (ctr=0; ctr < numVertices; ctr++) {
    v = findNewVertexOfDegreeZero(); O(V)
    put v next in output
    for each w adjacent to v
        w.indegree--; 
}
```

You can use a helper variable \( d \) – the out degree of a vertex
Topological Sort: Running time?

labelEachVertexWithItsInDegree(); \[ O(V+E) \]
for(ctr=0; ctr < numVertices; ctr++) { \[ V \text{ times} \]
    v = findNewVertexOfDegreeZero(); \[ O(V) \]
    put v next in output \[ O(1) \]
    for each w adjacent to v
        w.indegree--;
}

You can use a helper variable \( d \) – the out degree of a vertex
Topological Sort: Running time?

```java
labelEachVertexWithItsInDegree(); O(V+E)
for(ctr=0; ctr < numVertices; ctr++){
    v = findNewVertexOfDegreeZero(); O(V)
    put v next in output O(1)
    for each w adjacent to v d times
        w.indegree--;
}
```

You can use a helper variable d – the out degree of a vertex
You can use a helper variable \( d \) – the out degree of a vertex
Topological Sort: Running time?

```java
labelEachVertexWithItsInDegree(); \quad O(V+E)
for ctr=0; ctr < numVertices; ctr++\{ \quad V \text{ times}
    v = findNewVertexOfDegreeZero(); \quad O(V)
    put v next in output \quad O(1)
    for each w adjacent to v \quad d \text{ times}
        w.indegree--; \quad O(1)
\}
```

\[ O(V + E + V^*(V + 1 + d)) \]

You can use a helper variable $d$ – the out degree of a vertex
Topological Sort: Running time?

```plaintext
labelEachVertexWithItsInDegree(); \quad O(V+E)
for(ctr=0; ctr < numVertices; ctr++) { \quad V times
    v = findNewVertexOfDegreeZero(); \quad O(V)
    put v next in output \quad O(1)
    for each w adjacent to v \quad d times
        w.indegree--; \quad O(1)
}
```

\[ O(V + E + V^2 + V + V^*d) \]

You can use a helper variable \( d \) – the out degree of a vertex
Topological Sort: Running time?

```
labelEachVertexWithItsInDegree(); O(V+E)
for(ctr=0; ctr < numVertices; ctr++){
    v = findNewVertexOfDegreeZero(); O(V)
    put v next in output O(1)
    for each w adjacent to v d times
        w.indegree--; O(1)
}
```

- $O(V^2 + E + V^*d)$
- $O(V + E + V^2 + V + V^*d))$
- $O(V + E + V^*(V + 1 + d))$

You can use a helper variable $d$ – the out degree of a vertex
Topological Sort: Running time?

```java
labelEachVertexWithItsInDegree(); O(V+E)
for(ctr=0; ctr < numVertices; ctr++){
    v = findNewVertexOfDegreeZero(); O(V)
    put v next in output O(1)
    for each w adjacent to v d times
        w.indegree--; O(1)
}
```

\[ O(V + E + V^*(V + 1 + d)) \]
\[ O(V + E + V^2 + V + V^*d) \]
\[ O(V^2 + E + E) \]

You can use a helper variable \( d \) – the out degree of a vertex
Topological Sort: Running time?

```
labelEachVertexWithItsInDegree(); O(V+E)
for(ctr=0; ctr < numVertices; ctr++) { V times
  v = findNewVertexOfDegreeZero(); O(V)
  put v next in output O(1)
  for each w adjacent to v d times
    w.indegree--; O(1)
}
```

\[ O(V + E + V^*(V + 1 + d)) \]
\[ O(V + E + V^2 + V + V^*d) \]
\[ O(V^2 + E + E) \]
\[ O(V^2 + E) \]

You can use a helper variable \( d \) – the out degree of a vertex
Topological Sort: Running time?

What is the worst-case running time?
- Initialization $O(|V| + |E|)$ (assuming adjacency list)
- Sum of all find-new-vertex $O(|V|^2)$ (because each $O(|V|)$)
- Sum of all decrements $O(|E|)$ (assuming adjacency list)
- So total is $O(|V|^2 + |E|)$ – not good for a sparse graph!

```java
labelEachVertexWithItsInDegree();
for(ctr=0; ctr < numVertices; ctr++){
    v = findNewVertexOfDegreeZero();
    put v next in output
    for each w adjacent to v
        w.indegree--;
}
```
Doing better

The trick is to avoid searching for a zero-degree node every time!
- Keep the “pending” zero-degree nodes in a list, stack, queue, box, table, or something
- Order we process them affects output but not correctness or efficiency provided add/remove are both $O(1)$

Using a queue:

1. Label each vertex with its in-degree, enqueue 0-degree nodes
2. While queue is not empty
   a) $v = \text{dequeue}()$
   b) Output $v$ and remove it from the graph
   c) For each vertex $w$ adjacent to $v$ (i.e. $w$ such that $(v,w)$ in $E$), decrement the in-degree of $w$, if new degree is 0, enqueue it
Topological Sort (optimized): Running time?

```java
labelAllAndEnqueueZeros();
for(ctr=0; ctr < numVertices; ctr++) {
    v = dequeue();
    put v next in output
    for each w adjacent to v {
        w.indegree--;
        if(w.indegree==0)
            enqueue(w);
    }
}
```
Topological Sort (optimized): Running time?

```java
labelAllAndEnqueueZeros();
for (ctr=0; ctr < numVertices; ctr++) {  \( O(V + E) \) V times
    v = dequeue();  \( O(1) \)
    put v next in output  \( O(1) \)
    for each w adjacent to v {  d times
        w.indegree--;  \( O(1) \)
        if (w.indegree == 0)  \( O(1) \)
            enqueue(w);  \( O(1) \)
    }
}
```

- What is the worst-case running time?
  - Initialization: \( O(|V|+|E|) \) (assuming adjacency list)
  - Sum of all enqueues and dequeues: \( O(|V|) \)
  - Sum of all decrements: \( O(|E|) \) (assuming adjacency list)
  - So total is \( O(|E| + |V|) \) – much better for sparse graph!
Graph Traversals

Next problem: For an arbitrary graph and a starting node $v$, find all nodes \textit{reachable} (i.e., there exists a path) from $v$
- Possibly “do something” for each node (an iterator!)
  - E.g. Print to output, set some field, etc.

Related Questions:
- Is an undirected graph connected?
- Is a directed graph weakly / strongly connected?
  - For strongly, need a cycle back to starting node

Basic idea:
- Keep following nodes
- But “mark” nodes after visiting them, so the traversal terminates and processes each reachable node exactly once
Graph Traversal: Abstract Idea

traverseGraph(Node start) {
  Set pending = emptySet();
  pending.add(start)
  mark start as visited
  while(pending is not empty) {
    next = pending.remove()
    for each node u adjacent to next
      if(u is not marked) {
        mark u
        pending.add(u)
      }
  }
}
Running time and options

- Assuming add and remove are \( O(1) \), entire traversal is \( O(|E|) \)
  - Use an adjacency list representation

- The order we traverse depends entirely on how add and remove work/are implemented
  - Depth-first graph search (DFS): a stack
  - Breadth-first graph search (BFS): a queue

- DFS and BFS are “big ideas” in computer science
  - Depth: recursively explore one part before going back to the other parts not yet explored
  - Breadth: Explore areas closer to the start node first
Recursive DFS, Example: trees

- A tree is a graph and DFS and BFS are particularly easy to “see”

```
DFS(Node start) {
    mark and “process” (e.g. print) start
    for each node u adjacent to start
        if u is not marked
            DFS(u)
}
```

Order processed: A, B, D, E, C, F, G, H
- Exactly what we called a “pre-order traversal” for trees
- The marking is not needed here, but we need it to support arbitrary graphs, we need a way to process each node exactly once
DFS with a stack, Example: trees

DFS2(Node start) {
    initialize stack s to hold start
    mark start as visited
    while(s is not empty) {
        next = s.pop() // and “process”
        for each node u adjacent to next
            if(u is not marked)
                mark u and push onto s
    }
}

Order processed:
• A different but perfectly fine traversal
DFS with a stack, Example: trees

DFS2(Node start) {
    initialize stack s to hold start
    mark start as visited
    while(s is not empty) {
        next = s.pop() // and “process”
        for each node u adjacent to next
            if(u is not marked)
                mark u and push onto s
    }
}

Order processed:
• A different but perfectly fine traversal
**DFS with a stack, Example: trees**

DFS2(Node start) {
    initialize stack s to hold start
    mark start as visited
    while(s is not empty) {
        next = s.pop() // and “process”
        for each node u adjacent to next
            if(u is not marked)
                mark u and push onto s
    }
}

Order processed: A
- A different but perfectly fine traversal
DFS with a stack, Example: trees

DFS2(Node start) {
    initialize stack s to hold start
    mark start as visited
    while(s is not empty) {
        next = s.pop() // and “process”
        for each node u adjacent to next
            if(u is not marked)
                mark u and push onto s
    }
}

Order processed: A
- A different but perfectly fine traversal
DFS with a stack, Example: trees

DFS2(Node start) {
    initialize stack s to hold start
    mark start as visited
    while(s is not empty) {
        next = s.pop() // and “process”
        for each node u adjacent to next
            if(u is not marked)
                mark u and push onto s
    }
}

Order processed: A, C
• A different but perfectly fine traversal
**DFS with a stack, Example: trees**

DFS2(Node start) {
    initialize stack s to hold start
    mark start as visited
    while(s is not empty) {
        next = s.pop() // and "process"
        for each node u adjacent to next
            if(u is not marked)
                mark u and push onto s
    }
}

Order processed: A, C, F
- A different but perfectly fine traversal
DFS with a stack, Example: trees

DFS2(Node start) {
    initialize stack s to hold start
    mark start as visited
    while(s is not empty) {
        next = s.pop() // and “process”
        for each node u adjacent to next
            if(u is not marked)
                mark u and push onto s
    }
}

Order processed: A, C, F

• A different but perfectly fine traversal
DFS with a stack, Example: trees

```java
DFS2(Node start) {
    initialize stack s to hold start
    mark start as visited
    while(s is not empty) {
        next = s.pop() // and "process"
        for each node u adjacent to next
            if(u is not marked)
                mark u and push onto s
    }
}
```

Order processed: A, C, F, H, G, B, E, D
- A different but perfectly fine traversal
**BFS with a queue, Example: trees**

BFS(Node start) {
  initialize queue q to hold start
  mark start as visited
  while(q is not empty) {
    next = q.dequeue() // and “process”
    for each node u adjacent to next
      if(u is not marked)
        mark u and enqueue onto q
  }
}

Order processed:
- A “level-order” traversal
**BFS with a queue, Example: trees**

BFS(Node start) {
    initialize queue q to hold start
    mark start as visited
    while(q is not empty) {
        next = q.dequeue() // and “process”
        for each node u adjacent to next
            if(u is not marked)
                mark u and enqueue onto q
    }
}

Order processed:

- A “level-order” traversal
BFS with a queue, Example: trees

BFS(Node start) {
  initialize queue q to hold start
  mark start as visited
  while(q is not empty) {
    next = q.dequeue() // and “process”
    for each node u adjacent to next
      if(u is not marked)
        mark u and enqueue onto q
  }
}

Order processed: A
• A “level-order” traversal
BFS with a queue, Example: trees

BFS(Node start) {
    initialize queue q to hold start
    mark start as visited
    while(q is not empty) {
        next = q.dequeue() // and “process”
        for each node u adjacent to next
            if(u is not marked)
                mark u and enqueue onto q
    }
}

Order processed: A
• A “level-order” traversal
BFS with a queue, Example: trees

BFS(Node start) {
    initialize queue q to hold start
    mark start as visited
    while(q is not empty) {
        next = q.dequeue() // and “process”
        for each node u adjacent to next
            if(u is not marked)
                mark u and enqueue onto q
    }
}

Order processed: A, B
• A “level-order” traversal
BFS with a queue, Example: trees

BFS(Node start) {
    initialize queue q to hold start
    mark start as visited
    while (q is not empty) {
        next = q.dequeue() // and “process”
        for each node u adjacent to next
            if (u is not marked)
                mark u and enqueue onto q
    }
}

Order processed: A, B
• A “level-order” traversal
BFS with a queue, Example: trees

BFS(Node start) {
    initialize queue q to hold start
    mark start as visited
    while(q is not empty) {
        next = q.dequeue() // and “process”
        for each node u adjacent to next
            if(u is not marked)
                mark u and enqueue onto q
    }
}

Order processed: A, B, C, D, E, F, G, H
• A “level-order” traversal
**DFS/BFS Comparison**

Breadth-first search:
- Always finds shortest paths, i.e., “optimal solutions
  - Better for “what is the shortest path from \(x\) to \(y\)”
- Queue may hold \(O(|V|)\) nodes (e.g. at the bottom level of binary tree of height \(h\), \(2^h\) nodes in queue)

Depth-first search:
- Can use less space in finding a path
  - If *longest path* in the graph is \(p\) and highest out-degree is \(d\) then DFS stack never has more than \(d*p\) elements

A third approach: *Iterative deepening (IDDFS)*:
- Try DFS but don’t allow recursion more than \(k\) levels deep.
  - If that fails, increment \(k\) and start the entire search over
- Like BFS, finds shortest paths. Like DFS, less space.
Saving the path

• Our graph traversals can answer the “reachability question”:
  – “Is there a path from node x to node y?”

• Q: But what if we want to output the actual path?
  – Like getting driving directions rather than just knowing it’s possible to get there!

• A: Like this:
  – Instead of just “marking” a node, store the previous node along the path (when processing u causes us to add v to the search, set v.path field to be u)
  – When you reach the goal, follow path fields backwards to where you started (and then reverse the answer)
  – If just wanted path length, could put the integer distance at each node instead
Example using BFS

What is a path from Seattle to Austin

- Remember marked nodes are not re-enqueued
- Note shortest paths may not be unique
Example using BFS

What is a path from Seattle to Austin
  – Remember marked nodes are not re-enqueued
  – Note shortest paths may not be unique