

TOPOLOGICAL SORTING

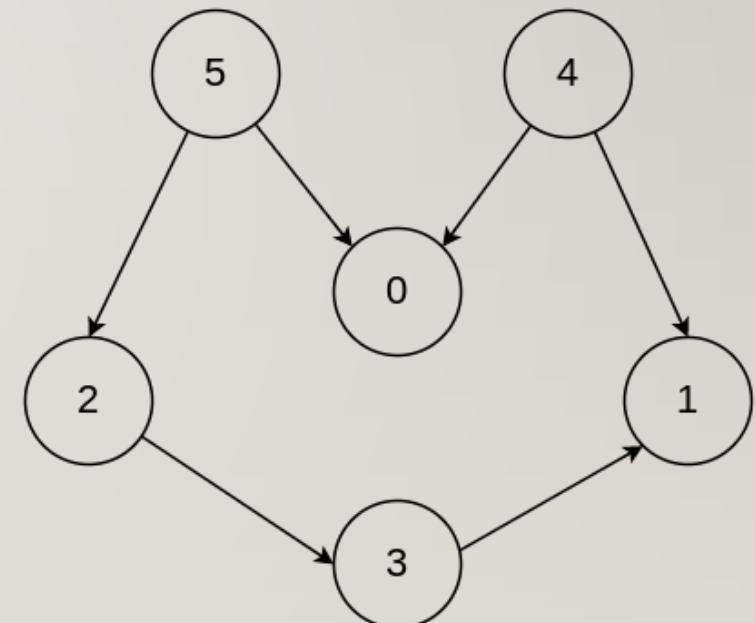
Ordering tasks with dependencies in Directed Acyclic Graphs (DAGs)

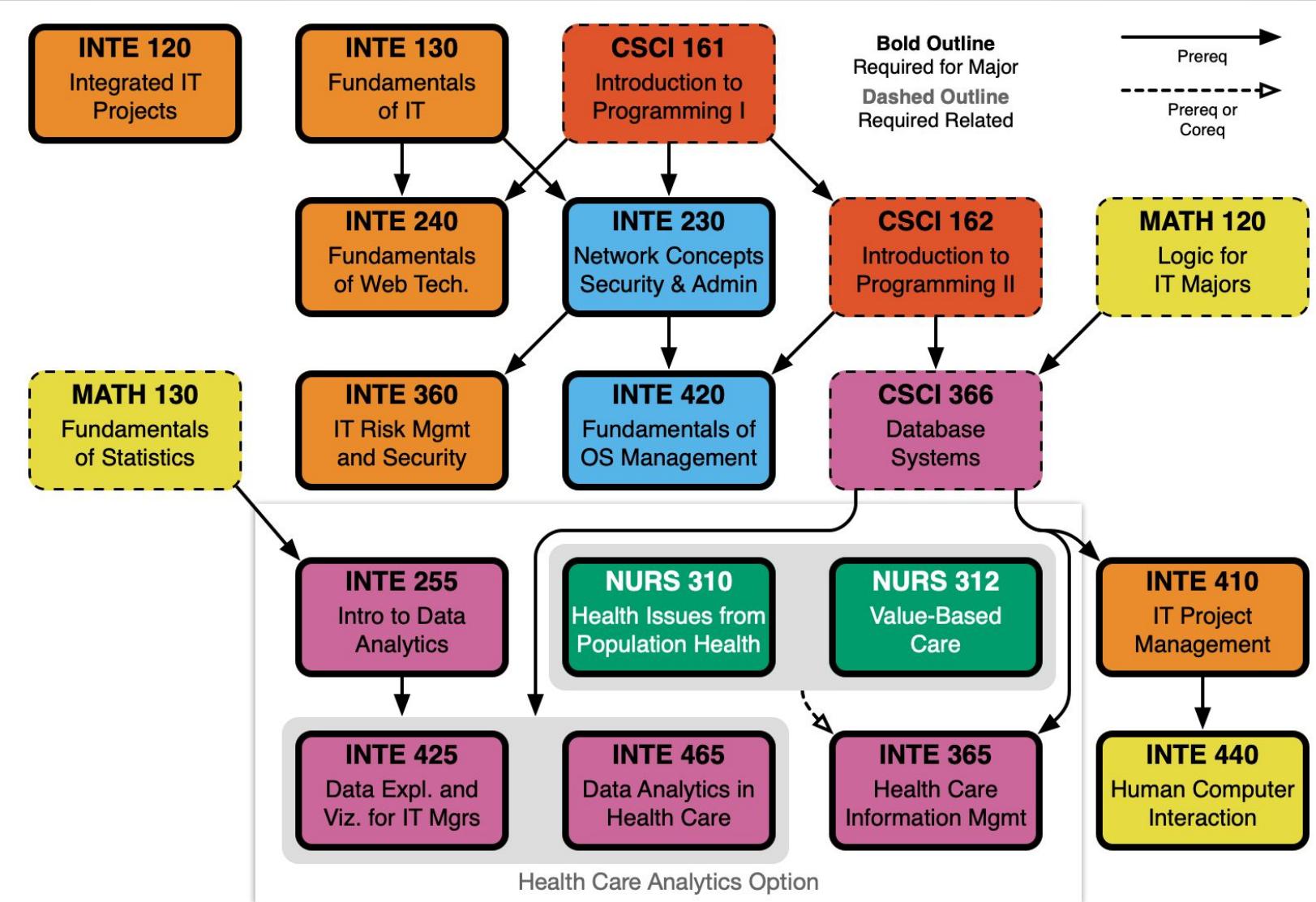
DIRECTED ACYCLIC GRAPH

A **Directed Acyclic Graph (DAG)** is a directed graph that does not contain any cycles.

Applications:

- Task scheduling (e.g., build systems, course prerequisites)
- Data processing pipelines
- Dependency Resolution (e.g., npm)
- Version control systems (e.g., Git commit history)
- Topological sorting algorithms



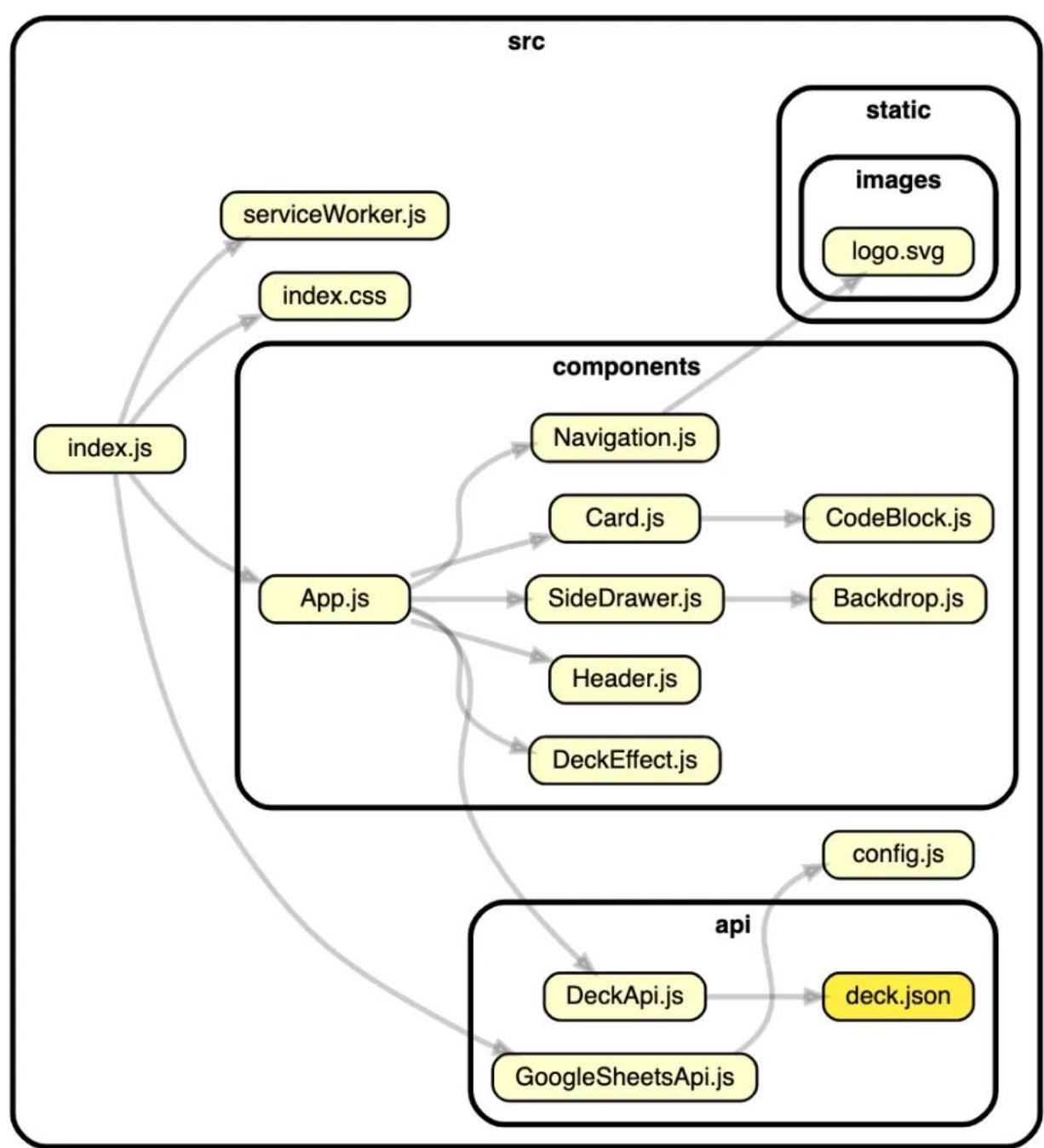


INTE 300/400
Internship
(1-6 cr)

INTE 486
Topics
(1-4 cr)

INTE 489/499
Honors Thesis
(1-4 cr)

INTE 498
Independent Study
(1-4 cr)



EXAMPLE PROBLEM

① Build Systems

IDEs like Eclipse, NetBeans must build projects with many interdependent libraries.

Topological Sort helps determine the order in which libraries should be built or included.

② Advanced Packaging Tool (APT)

In Linux, apt-get installs software with dependencies.

Topological Sort ensures packages are installed in the correct dependency order.

EXAMPLE PROBLEM

③ Task Scheduling

Useful for scheduling tasks with dependencies. Helps determine the correct sequence of task execution.

④ Prerequisite Problems

Common in education or workflows: some tasks require others to be done first.

Example: must complete Basic Algorithms before Advanced Algorithms.

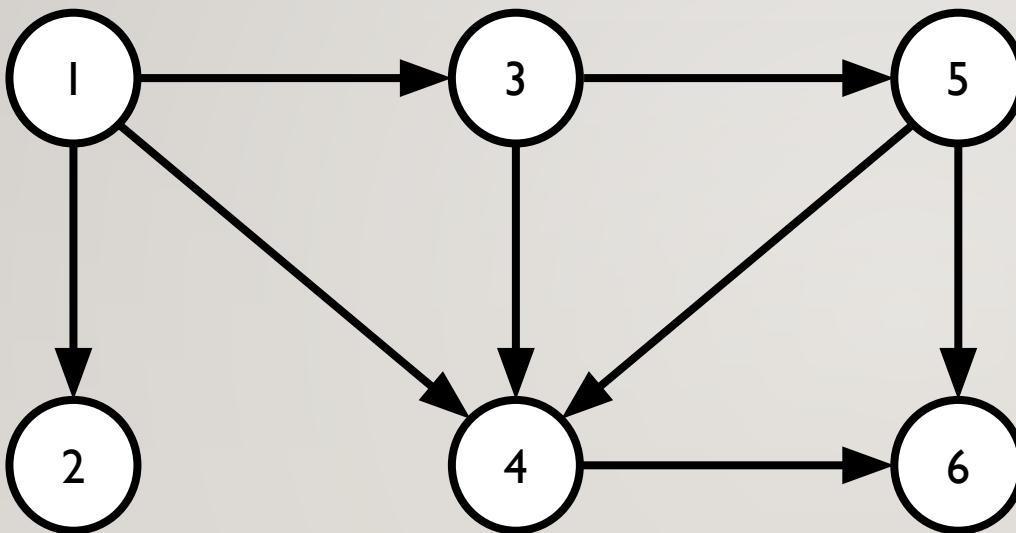
TOPOLOGICAL SORTING

A topological sort of a **Directed Acyclic Graph (DAG)** is a linear ordering of vertices such that for every directed edge $u-v$, vertex **u** comes before **v** in the ordering.

Every finite DAG has a topological sort.

Note: Topological Sorting for a graph is not possible if the graph is not a [**DAG**](#).

TOPOLOGICAL SORTING



$$T = (1, 3, 2, 5, 4, 6)$$

Note: there may be multiple topological orderings.

$T = (1, 2, 3, 5, 4, 6)$ is also valid.

TOPOLOGICAL SORT ALGORITHMS

Two algorithms for finding the topological order of a graph:

1. Kahn's Algorithm
2. DFS

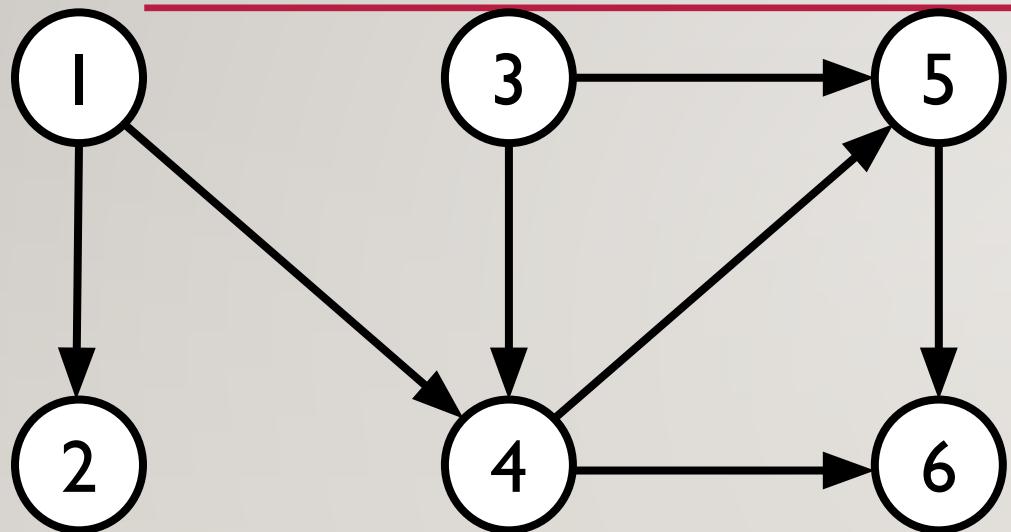
KAHN'S ALGORITHM

Kahn's Algorithm works by repeatedly finding vertices with *no incoming edges*, removing them from the graph, and *updating the incoming edges* of the vertices connected from the removed edges. This process continues until all vertices have been ordered.

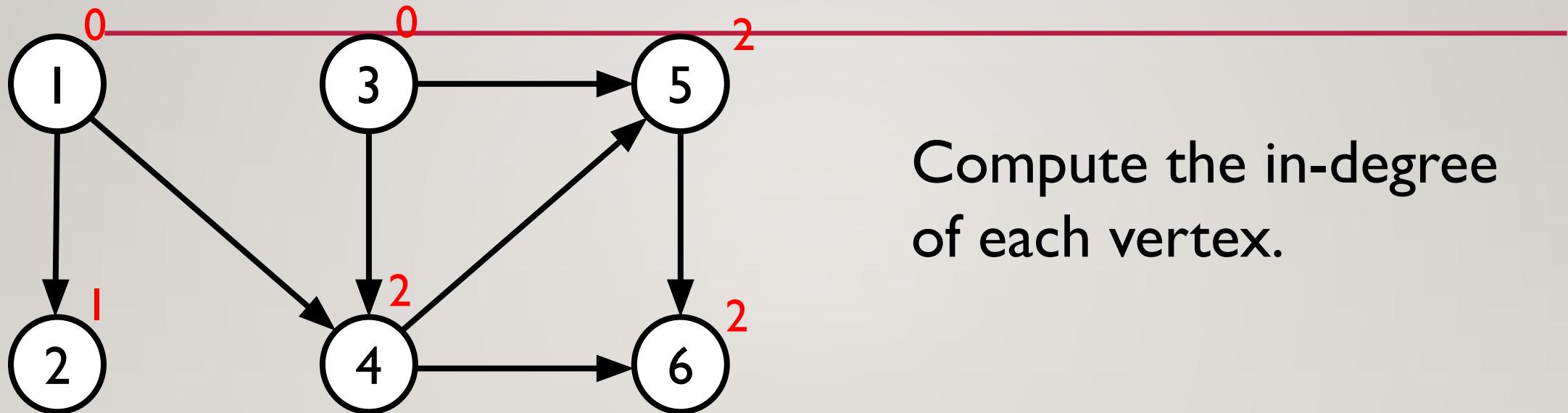
KAHN'S ALGORITHM

1. Compute the in-degree of each vertex
2. Add all of the vertices with an in-degree of 0 onto a queue Q
3. Initialize an empty list `topo_order`
4. While Q is not empty:
 - a) Remove a node v from Q and add it to `topo_order`
 - b) For each outgoing edge from v , decrement the in-degree of the destination node w by 1.
 - c) If the in-degree of w becomes 0, add w to the queue.
5. If the queue is empty and there are still nodes in the graph, the graph contains a cycle and cannot be topologically sorted.
6. `topo_order` represent the topological ordering of the graph.

KAHN'S ALGORITHM

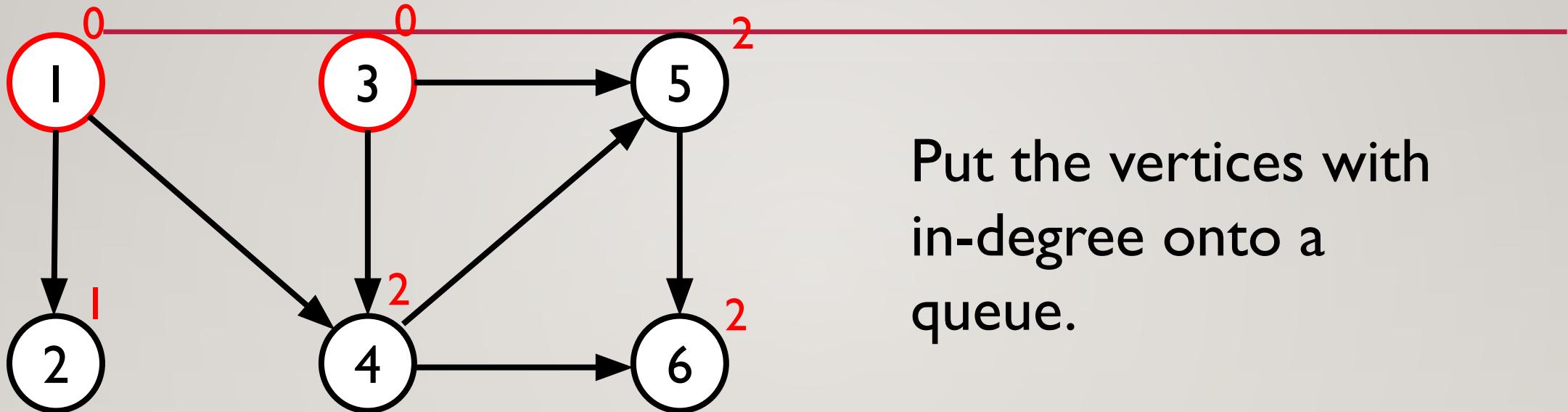


KAHN'S ALGORITHM



Compute the in-degree
of each vertex.

KAHN'S ALGORITHM

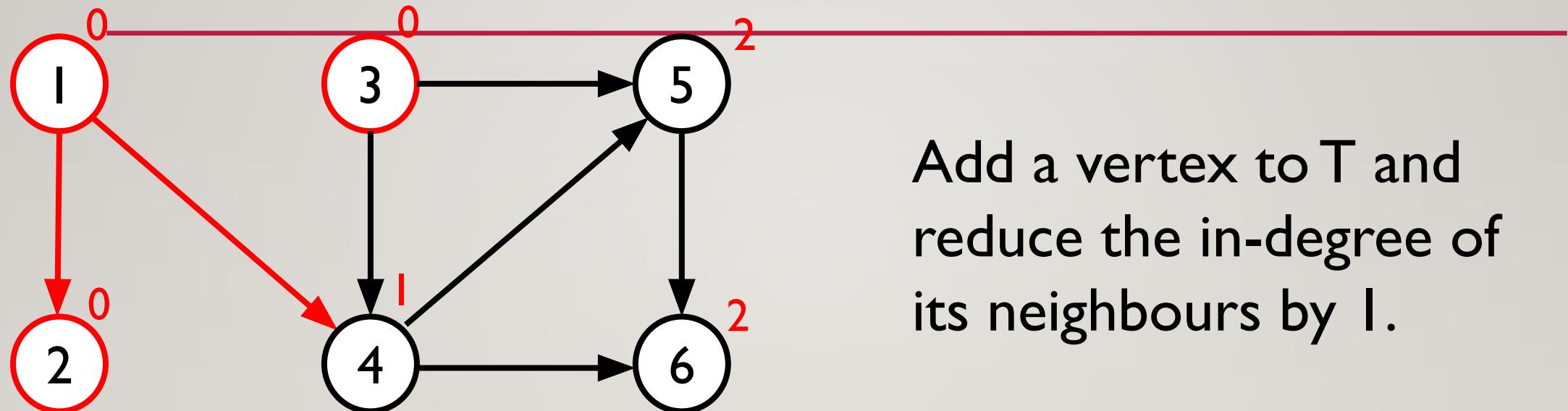


Put the vertices with
in-degree onto a
queue.

$$Q = (1, 3)$$

$$T = ()$$

KAHN'S ALGORITHM

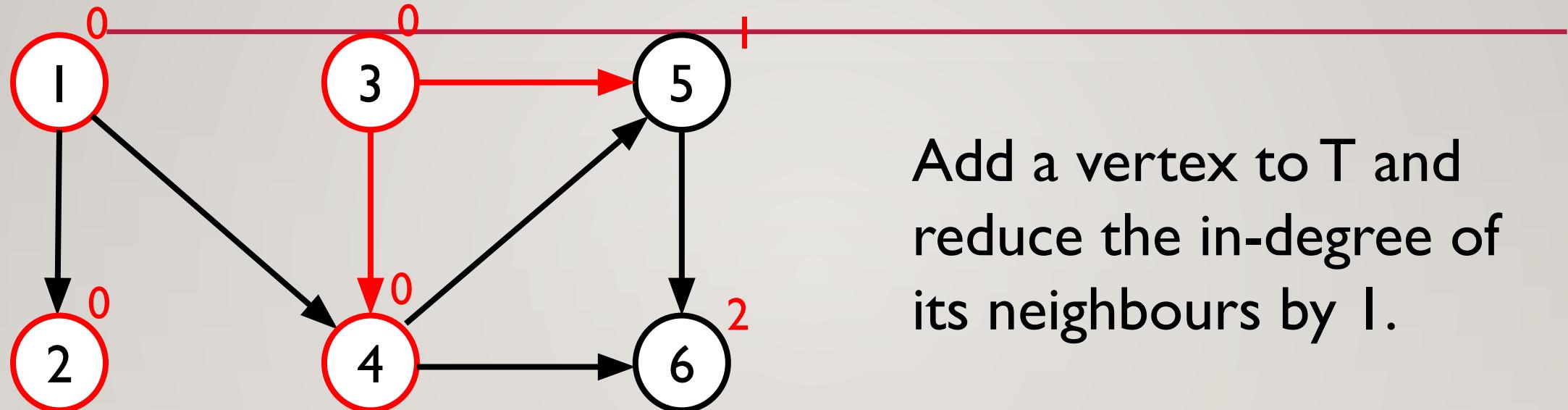


Add a vertex to T and reduce the in-degree of its neighbours by 1.

$$Q = (3, 2)$$

$$T = (1)$$

KAHN'S ALGORITHM

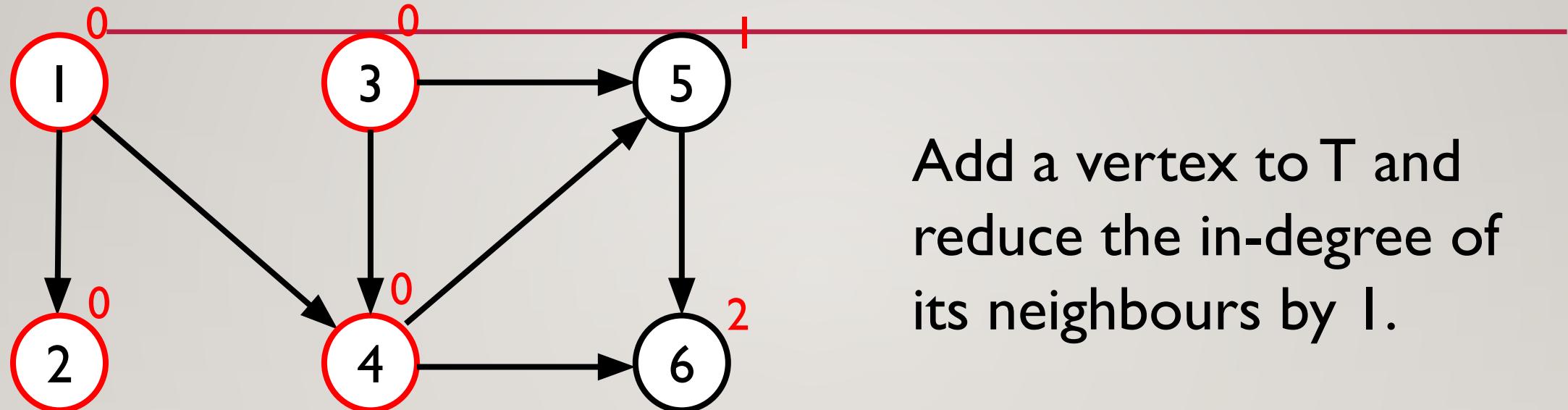


Add a vertex to T and reduce the in-degree of its neighbours by 1.

$$Q = (2, 4)$$

$$T = (1, 3)$$

KAHN'S ALGORITHM

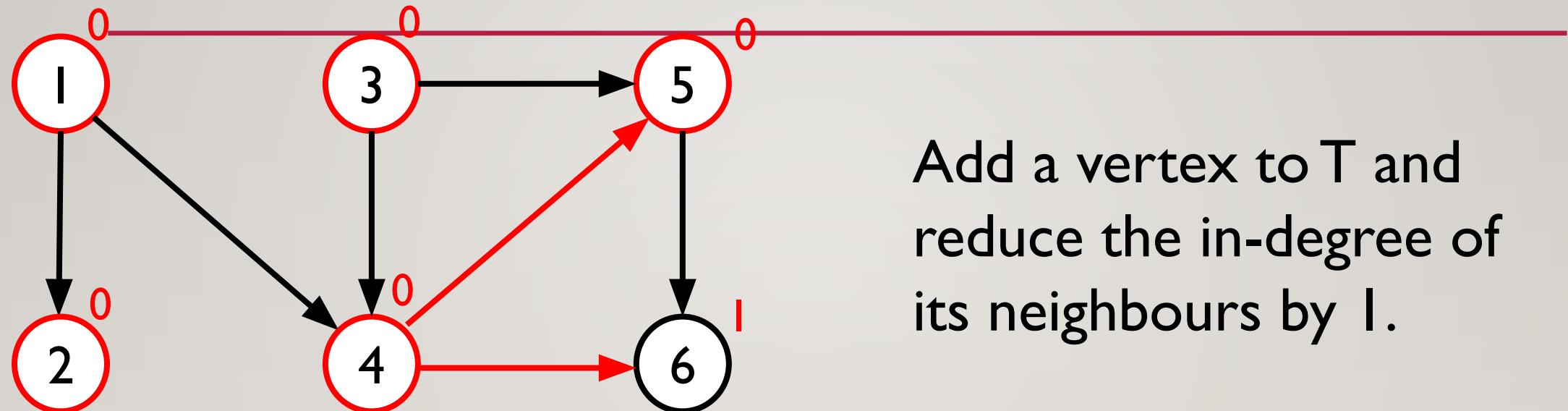


Add a vertex to T and reduce the in-degree of its neighbours by 1.

$$Q = (4)$$

$$T = (1, 3, 2)$$

KAHN'S ALGORITHM

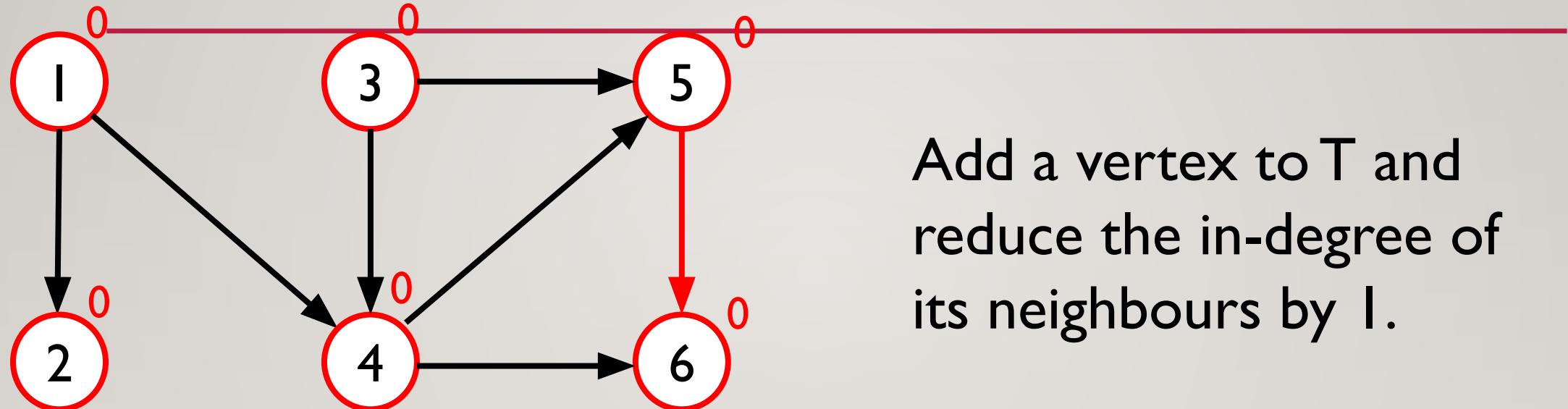


Add a vertex to T and reduce the in-degree of its neighbours by 1.

$$Q = (5)$$

$$T = (1, 3, 2, 4)$$

KAHN'S ALGORITHM

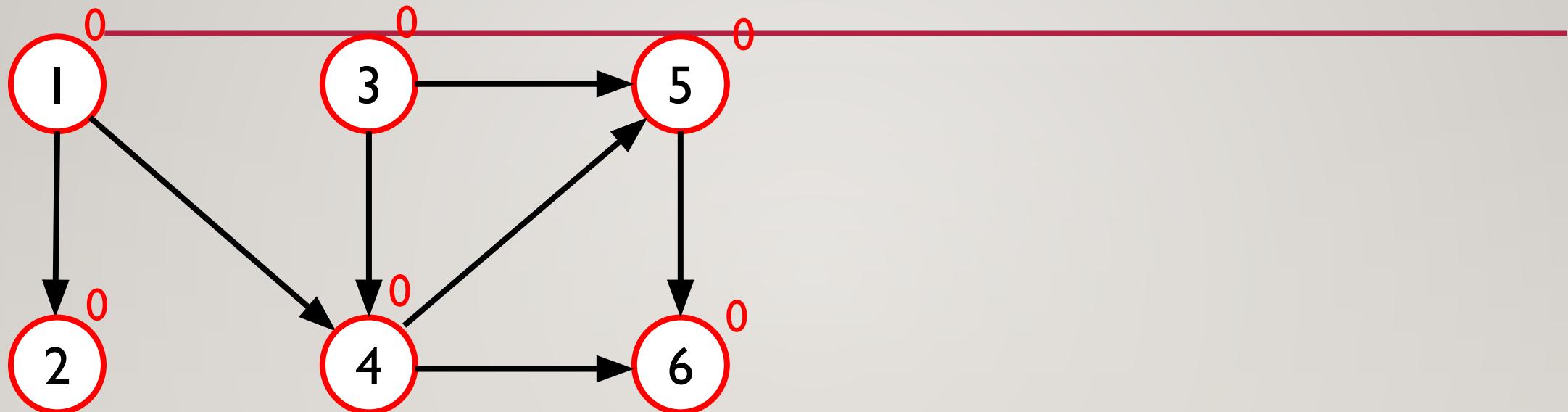


Add a vertex to T and reduce the in-degree of its neighbours by 1.

$$Q = (6)$$

$$T = (1, 3, 2, 4, 5)$$

KAHN'S ALGORITHM



$$Q = ()$$

$$T = (1, 3, 2, 4, 5, 6)$$

KAHN'S ALGORITHM

Time & space

- Time: $O(|V| + |E|)$
- Space: $O(|V| + |E|)$ (adj list + queue)

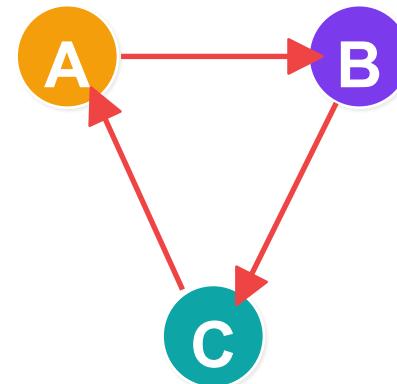
Why cycle \Rightarrow failure?

In a directed cycle, every node has in-degree ≥ 1 (from within the cycle).

So the queue becomes empty while nodes remain.

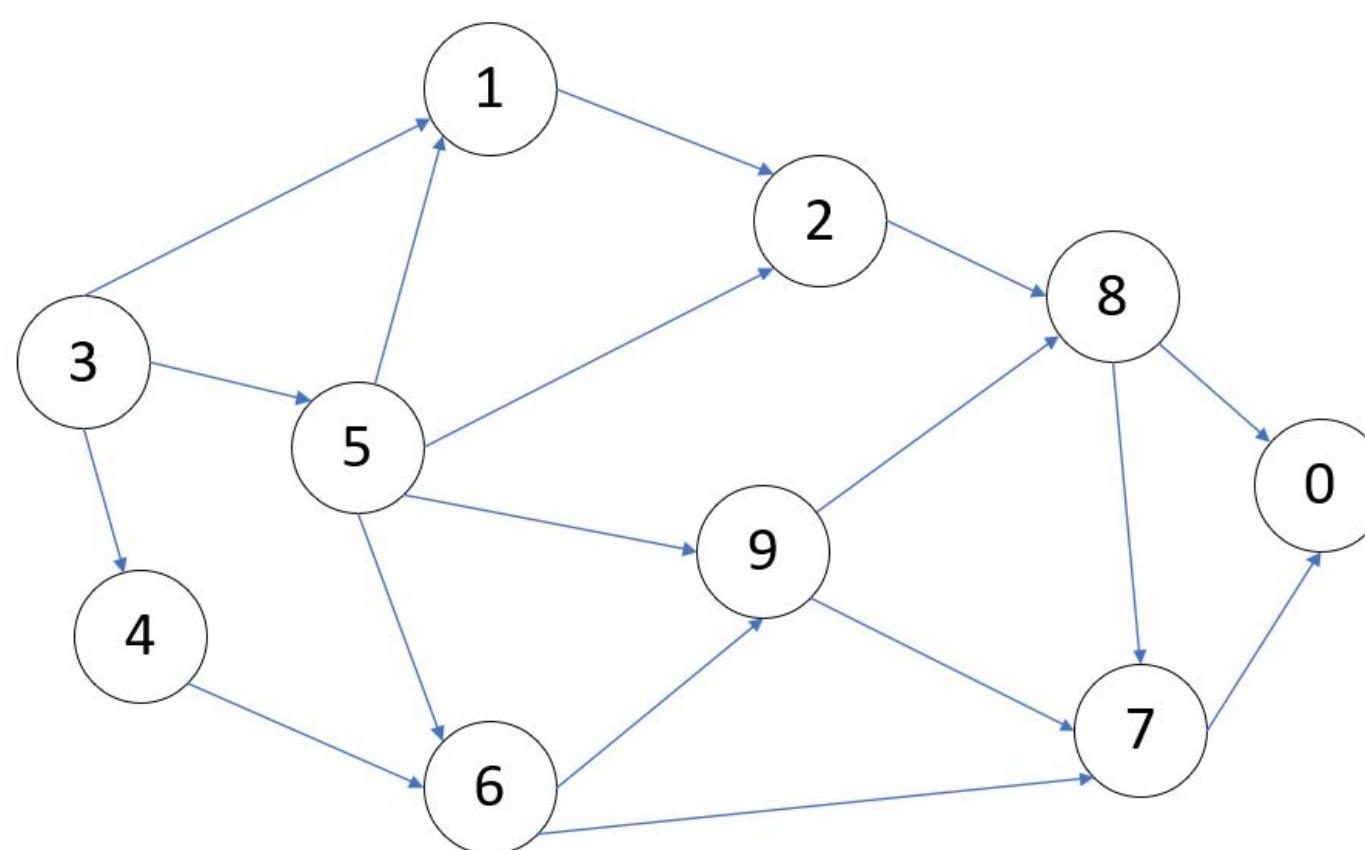
Small counterexample

Edges: A \rightarrow B, B \rightarrow C, C \rightarrow A



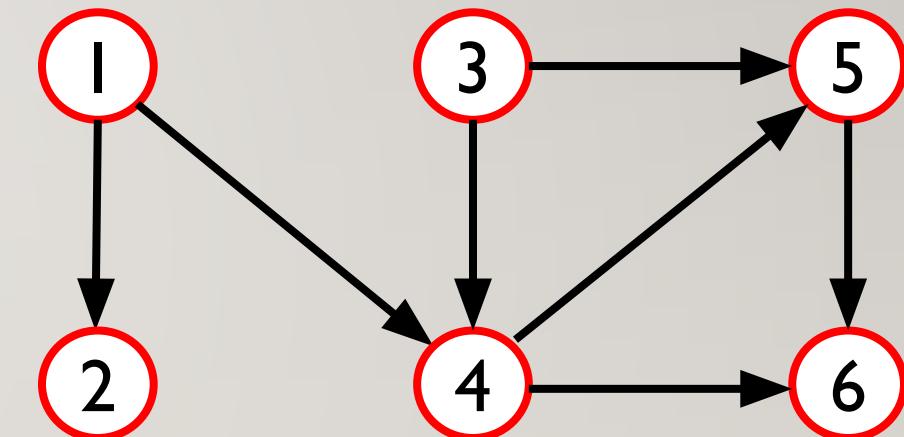
All in-degrees = 1 \rightarrow Q is empty at start.

EXAMPLES



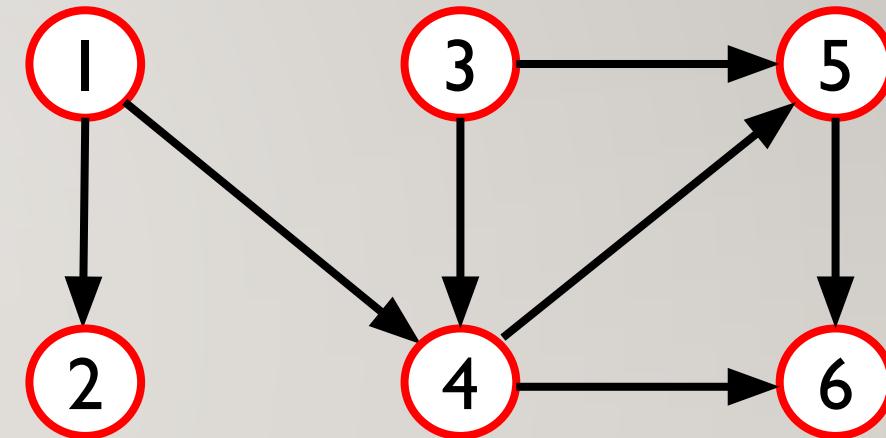
TOPOLOGICAL SORT USING DFS

- Initialize an empty list *topo_order* = []
- For each unvisited vertex in the graph, do the following:
 - Find the **post ordering** of DFS with the unvisted vertex
 - Append the post ordering DFS to *topo_order*
- Reverse *topo_order* → that is the topological order



TOPOLOGICAL SORT USING DFS

1. Initialize an empty list: `topo_order = []`
2. For each unvisited vertex u in the graph:
 - a. Perform $\text{DFS}(u)$:
 - i. Mark u as visited
 - ii. Recursively DFS all neighbors v of u that are unvisited
 - iii. After all neighbors are visited, append u to `topo_order`
3. Reverse `topo_order` → that is the topological order

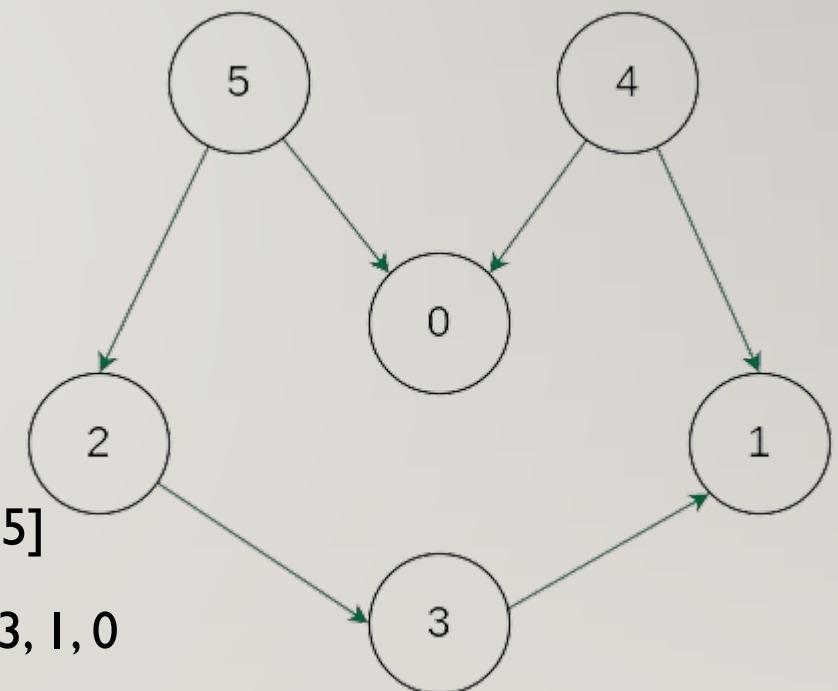


TOPOLOGICAL SORT USING DFS

Initialize: topo_order = []

- Vertex 0 -> postordering is [0] -> topo_order=[0]
- Vertex 1 -> postordering is [1] -> topo_order=[0, 1]
- Vertex 2 -> postordering is [3, 2] -> topo_order=[0, 1, 3, 2]
- Vertex 4 -> postordering is [4] -> topo_order=[0, 1, 3, 2, 4]
- Vertex 5 -> postordering is [5] -> topo_order=[0, 1, 3, 2, 4, 5]

So, the topological order is reverse of [0, 1, 3, 2, 4, 5] = 5, 4, 2, 3, 1, 0



TOPOLOGICAL SORT USING DFS

- Time complexity: $O(|V| + |E|)$
- Space complexity: $O(|V| + |E|)$

LEXICOGRAPHICALLY SMALLEST TOPOLOGICAL ORDERING

- Lexicographically smallest topological ordering means that if two vertices in a graph do not have any incoming edge then the vertex with the smaller number should appear first in the ordering.
- How?

LEXICOGRAPHICALLY SMALLEST TOPOLOGICAL ORDERING

Approach: Kahn's algorithm + Priority Queue

Time complexity: $O(|V|\log |V| + |E|)$

REFERENCES

- Kahn's Algorithm
- Topological Sorting
- Lexicographically Smallest Topological Ordering:
<https://www.geeksforgeeks.org/dsa/lexicographically-smallest-topological-ordering/>