Topic Introduction

Today, we're diving into the exciting world of **topological sorting** in graphs. This is a powerful algorithmic concept that's a staple in coding interviews, especially when you're dealing with dependencies or prerequisites. Whether you're building a software build system or working out the order of tasks in a project, topological sorting helps you arrange tasks so that every prerequisite comes before the task that depends on it.

What is Topological Sorting?
A topological sort is an ordering of the nodes in a **Directed Acyclic Graph (DAG)** such that for every directed edge from node A to node B, A comes before B in the ordering. If the graph has cycles, a topological sort is impossible—since you'd have circular dependencies.

How does it work?
There are two classic approaches:

- **Depth-First Search (DFS):** You traverse each node, marking nodes as visited, and add them to the sort order only after visiting all their neighbors.
- **Kahn's Algorithm (BFS-based):** You start with nodes that have no incoming edges (in-degree 0), add them to the ordering, and remove their outgoing edges, updating the in-degree of their neighbors.

Why is it useful in interviews?
Topological sorting is a go-to technique for problems involving dependency resolution, task scheduling, build order, or cycle detection in graphs. Interviewers love these problems because they test your understanding of graphs, cycles, and algorithmic thinking.

Simple Example (not from our target problems):
Suppose you have three tasks:

- Task 1 must be done before Task 2.
- Task 2 must be done before Task 3.

The graph is: 1 → 2 → 3.
A topological sort would be [1, 2, 3].

Today, we'll tackle three classic problems where topological sorting (and its related concepts) are the stars:

- Course Schedule
- Course Schedule II
- Minimum Height Trees

Why are these grouped together?
All three ask you to reason about graphs and dependencies. Course Schedule is about detecting cycles (can you finish all courses?). Course Schedule II is about finding a valid order. Minimum Height Trees flips the concept, asking you to find the best root(s) for a tree, which involves a clever twist on topological pruning.

Let’s take them step by step!

Problem 1: Course Schedule ([Leetcode #207](https://leetcode.com/problems/course-schedule/))

Problem, in plain English:

You're given `numCourses` labeled from 0 to `numCourses-1` and a list of prerequisite pairs: `[a, b]` means you must take course `b` before `a`.

Question: Can you finish all courses? (Is there a way to take courses without running into impossible prerequisites?)

Example:

Input:

```
numCourses = 2
prerequisites = [[1,0]]
```

Output: `True`

Explanation: Take course 0 first, then course 1.

Another Input:

```
numCourses = 2
prerequisites = [[1,0],[0,1]]
```

Output: `False`

Explanation: You can't take course 0 before 1 and 1 before 0—it's a cycle!

How should you approach this?

This is a classic **cycle detection** problem in a directed graph. If there's a cycle, you can't finish all courses. If there isn't a cycle, you can.

Try this on paper:

Imagine 3 courses and prerequisites: [[1,0],[2,1],[0,2]]. Can you finish all courses?

Brute-force approach:

Try all possible orders of courses (factorial time). Clearly infeasible for large `numCourses` (O(N!)).

Optimal approach:

Use **topological sorting**! If you can sort all courses, there's no cycle. If you can't, there is one.

We'll use **Kahn's Algorithm** here (BFS):

Step-by-step logic

- Build an adjacency list for the graph.
- Compute the in-degree (number of prerequisites) for each course.
- Start with all courses that have 0 in-degree (no prerequisites).
- Remove these from the graph, decrease in-degree for their neighbors.
- If at the end you've processed all courses, return `True`. If not, there's a cycle.

Python Solution:

```
from collections import deque, defaultdict

def canFinish(numCourses, prerequisites):
    # Step 1: Build the graph and compute in-degree
    graph = defaultdict(list)
    in_degree = [0] * numCourses

    for dest, src in prerequisites:
        graph[src].append(dest)
        in_degree[dest] += 1

    # Step 2: Collect courses with no prerequisites
    queue = deque([i for i in range(numCourses) if in_degree[i] == 0])
    taken = 0

    # Step 3: BFS
    while queue:
        course = queue.popleft()
        taken += 1  # We can take this course
        for neighbor in graph[course]:
            in_degree[neighbor] -= 1  # Remove dependency
            if in_degree[neighbor] == 0:
                queue.append(neighbor)

    # Step 4: If we've taken all courses, it's possible
    return taken == numCourses
```

Time Complexity: O(N + E)

- N = number of courses (nodes)
- E = number of prerequisite pairs (edges)

Space Complexity: O(N + E)

- For the adjacency list and in-degree array.

What does each part do?

- `graph`: Maps each course to the list of courses that depend on it.
- `in_degree`: Counts prerequisites for each course.
- `queue`: Courses we can take now (no prerequisites left).
- Main loop: Each time we "take" a course, we reduce dependencies for its neighbors.

Trace Example:

Input: `numCourses = 4`, `prerequisites = [[1,0],[2,1],[3,2]]`

- Build graph: $0 \to 1 \to 2 \to 3$

- Start with course 0 (no prereqs)
- Take 0, now 1 has 0 prereqs
- Take 1, now 2 has 0 prereqs
- Take 2, now 3 has 0 prereqs
- Take 3, done! All courses taken.

Try this test case yourself:
`numCourses = 3`, `prerequisites = [[0,1],[1,2],[2,0]]`
(Can you finish all courses?)

Take a moment to solve this on your own before jumping into the solution!

Reflective prompt:
Did you know you could also use DFS with cycle detection for this? Give that a shot after finishing this!

Problem 2: Course Schedule II ([Leetcode #210](https://leetcode.com/problems/course-schedule-ii/))

How is this different?
Instead of just checking if you can finish all courses, now you must **return one possible order** in which to take all the courses. If it's impossible (there's a cycle), return an empty list.

Example:
Input:

```
numCourses = 4
prerequisites = [[1,0],[2,0],[3,1],[3,2]]
```

Output: `[0,1,2,3]` or `[0,2,1,3]`
Explanation:

- Take 0 first (no prereqs)
- Then 1 and 2 (both depend only on 0)
- Then 3 (depends on both 1 and 2)

Try this test case:
`numCourses = 2`, `prerequisites = [[1,0],[0,1]]`
What should the output be?

Brute-force:
Generate all possible orders and check validity. Again, factorial time.

Optimal approach:
Use **topological sorting**, just like before! This time, we'll record the order as we process courses.

Step-by-step breakdown:

- Build the graph and compute in-degree.

- Start with courses with in-degree 0.
- For each course taken, append it to the result list, decrement in-degree for neighbors.
- If you process all courses, return the order. If not, return an empty list.

Pseudocode:

```
function findOrder(numCourses, prerequisites):
    graph = adjacency list from prerequisites
    in_degree = array of zeros

    for each (dest, src) in prerequisites:
        graph[src].append(dest)
        in_degree[dest] += 1

    queue = all courses with in_degree 0
    result = []

    while queue not empty:
        course = queue.pop()
        result.append(course)
        for neighbor in graph[course]:
            in_degree[neighbor] -= 1
            if in_degree[neighbor] == 0:
                queue.append(neighbor)

    if length of result == numCourses:
        return result
    else:
        return []
```

Example Trace:

`numCourses = 3`, `prerequisites = [[1,0],[2,0],[2,1]]`

- Graph: 0 → 1, 0 → 2, 1 → 2
- In-degree: [0,1,2]
- Queue: [0]
- Take 0: result = [0], update in-degree: 1 (0), 2 (1)
- Queue: [1]
- Take 1: result = [0,1], update in-degree: 2 (0)
- Queue: [2]
- Take 2: result = [0,1,2]
- All courses taken!

Try this yourself:

`numCourses = 4`, `prerequisites = [[1,0],[2,1],[3,2],[1,3]]`

Time Complexity: O(N + E)

Space Complexity: O(N + E)

Problem 3: Minimum Height Trees ([Leetcode #310](https://leetcode.com/problems/minimum-height-trees/))

What's different or more challenging here?

You’re given an **undirected tree** with `n` nodes (0 to n-1).

Your task: Find all the root nodes that would result in a **minimum height tree** when the tree is rooted at that node.

Example:

Input:

```
n = 4
edges = [[1,0],[1,2],[1,3]]
```

Output: `[1]`

Explanation:

If you root the tree at node 1, the height is minimized.

Another test case:

`n = 6`, `edges = [[0,3],[1,3],[2,3],[4,3],[5,4]]`

What do you think the output should be?

Brute-force:

Try rooting the tree at every node, compute the height for each—very expensive (O(n^2)).

Optimal approach:

Use a **topological pruning** technique:

- Imagine peeling the tree layer by layer from the leaves (nodes with degree 1).
- The last node(s) left are the centroid(s), which minimize the tree height.

Step-by-step logic:

- Build the adjacency list.
- Identify all leaves (nodes with only one neighbor).
- While more than two nodes remain:
    - Remove leaves.
    - Update adjacency for neighbors.
    - Identify new leaves.
- Return the remaining node(s).

Pseudocode:

```
function findMinHeightTrees(n, edges):
    if n <= 2:
        return list of all nodes

    graph = adjacency list
    for each edge (u, v):
        graph[u].append(v)
        graph[v].append(u)

    leaves = all nodes with degree 1

    while n > 2:
        n -= length of leaves
        new_leaves = []
        for leaf in leaves:
            neighbor = only neighbor of leaf
            remove leaf from neighbor's adjacency
            if neighbor now has degree 1:
                new_leaves.append(neighbor)
        leaves = new_leaves

    return leaves
```

Example Trace:

Given: `n = 6`, `edges = [[0,3],[1,3],[2,3],[4,3],[5,4]]`

- Leaves: [0,1,2,5]
- Remove: [0,1,2,5] => new leaves: [4]
- Remove: [4] => new leaves: [3]
- Remaining nodes: [3]
- Output: [3]

Try this yourself:

`n = 7`, `edges = [[0,1],[1,2],[1,3],[2,4],[3,5],[4,6]]`

Time Complexity: O(N)

- Each node is only processed once.

Space Complexity: O(N)

Reflect:

Notice the similarity to topological sorting? Here, we're repeatedly removing "leaves" (like nodes with in-degree 0), but from an undirected graph.

Could this technique be used for other types of tree problems?

Summary and Next Steps

Today, you explored how topological sorting and its relatives help solve a variety of graph problems:

- Detecting cycles and feasibility (Course Schedule)
- Finding a valid order for tasks (Course Schedule II)

- Identifying optimal root(s) in a tree using a topological pruning approach (Minimum Height Trees)

Key patterns to remember:

- Build the graph (adjacency list)
- Track in-degrees or degrees
- Use a queue to process nodes in order
- Removing nodes with certain properties (like in-degree 0 or degree 1) can simplify the graph and reveal structure

Common mistakes:

- Forgetting to decrement in-degree/degree after removing a node
- Not handling disconnected graphs or edge cases (e.g., n <= 2)
- Confusing directed and undirected graph logic

Action List:

- Solve all 3 problems on your own—even the one with code above.
- Try solving Problem 2 and 3 using DFS instead of BFS, or vice versa.
- Explore other problems using topological sort (e.g., Alien Dictionary, Sequence Reconstruction).
- Compare your solution with others, focusing on edge cases and code clarity.
- Don’t worry if you get stuck! The key is steady practice and reflection.

Happy coding, and may your graphs always be acyclic (unless you need a challenge)!