Topic Introduction

Today's theme: Topological Sorting for Course Prerequisites

If you've ever tried to plan your university classes, you know you can't take "Advanced Algorithms" before "Intro to Programming". This kind of dependency is everywhere in computer science, and it's why *topological sorting* exists.

What is Topological Sorting?

Topological sorting is a way to order 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. Think of it as finding a sequence to do tasks without violating any "must do X before Y" rules.

- **How it works:**
  - It only applies to DAGs (no cycles allowed).
  - You look for nodes with no incoming edges (prerequisites).
  - Remove them, add to the order, and repeat for the rest.

Why is it useful in interviews?

- **Dependency resolution:** Scheduling courses, compiling code, task scheduling, and even some project management tools use this.
- **Cycle detection:** You can quickly tell if a set of dependencies is impossible.
- **Core skill:** Shows you understand graphs, BFS/DFS, and real-world modeling.

Simple Example:
Suppose you want to bake a cake:

- You must "buy ingredients" before you can "mix batter".
- You must "mix batter" before "bake cake".
- You must "bake cake" before "frost cake".

A valid order:

```
buy ingredients -> mix batter -> bake cake -> frost cake
```

Topological sorting gives you such valid sequences.

Why group these three problems?

- **Course Schedule**, **Course Schedule II**, and **Parallel Courses** all involve resolving course prerequisites.
  - *Course Schedule* asks: "Can I finish all my courses?" (cycle detection).
  - *Course Schedule II* asks: "Give me a valid order to finish all courses."
  - *Parallel Courses* asks: "What's the minimum number of semesters if I can take any number of courses at once, as long as prerequisites are met?"

All three use **topological sorting** to untangle dependencies!

Problem 1: Course Schedule ([LeetCode Link](https://leetcode.com/problems/course-schedule/))

Rephrased Problem Statement:

Given `numCourses` labeled from `0` to `numCourses-1`, and a list of prerequisite pairs `[a, b]` meaning you must take course `b` before course `a`, can you finish all courses? (Is it possible to complete all courses, given the prerequisites?)

Example Input/Output:

- Input: `numCourses = 2`, `prerequisites = [[1, 0]]`
- Output: `True`
- Reason: You can take course 0, then course 1.

Another Example:

- Input: `numCourses = 2`, `prerequisites = [[1, 0], [0, 1]]`
- Output: `False`
- Reason: There is a cycle: 0 needs 1, and 1 needs 0.

Thought Process:

- If there's any cycle in the dependency graph, some courses can't be finished.
- If the graph is a DAG (no cycles), it's possible.

Take a moment: Try drawing the example graphs on paper—see how cycles make finishing impossible!

Extra Test Case:

- Input: `numCourses = 3`, `prerequisites = [[1,0],[2,1],[0,2]]`
- Output: `False` (cycle: 0->1->2->0)

Brute Force Approach:

- Try all possible course orders to see if one works.
- Time complexity: O(N!) — not practical for even small N!

Optimal Approach: Kahn's Algorithm (BFS Topological Sort)

- Build a graph of courses.
- For each course, count incoming edges (prerequisites).
- Start with courses with 0 prerequisites.
- Remove them from the graph, reduce the in-degree of dependent courses.
- If you can remove all courses, it's possible (no cycle).
- If some remain, there's a cycle.

Python Solution

```
from collections import deque, defaultdict

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

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

    # Start with all courses that have no prerequisites
    queue = deque([i for i in range(numCourses) if in_degree[i] == 0])
    visited_count = 0

    while queue:
        course = queue.popleft()
        visited_count += 1
        for neighbor in graph[course]:
            in_degree[neighbor] -= 1  # Remove the prerequisite
            if in_degree[neighbor] == 0:
                queue.append(neighbor)

    # If we've visited all courses, it's possible
    return visited_count == numCourses
```

Time Complexity:

- O(N + E), where N is the number of courses and E is the number of prerequisite pairs.

Space Complexity:

- O(N + E), for the graph and in-degree array.

Explanation:

- `graph` holds the list of courses dependent on each course.
- `in_degree` counts how many prerequisites each course has.
- We put all courses with zero prerequisites into a queue.
- For each course removed from the queue, we "complete" it and update the prerequisites for its dependents.
- If we end up processing all courses (`visited_count == numCourses`), there is no cycle.

Trace Example:

Let's trace with:
`numCourses = 4`, `prerequisites = [[1,0],[2,1],[3,2]]`

- in_degree = [0,1,1,1]

- queue = [0]
- Visit 0: queue = [1] (decreased in_degree[1])
- Visit 1: queue = [2] (decreased in_degree[2])
- Visit 2: queue = [3] (decreased in_degree[3])
- Visit 3: queue = []
- visited_count = 4 -> return True

Try this test case:

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

(There is a cycle! Dry-run it.)

Self-attempt encouragement:

Pause here! Try implementing this yourself or dry-running with pen and paper to see how the queue evolves. It really cements the pattern.

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

What's different?

Now, instead of just asking "Can you finish?", you're asked to **return a valid order** of courses you can take, or an empty array if impossible.

Brute Force:

Try all permutations to find a valid one—totally impractical (O(N!)).

Optimal Approach:

We can use the same topological sort pattern as before, but now we **record the order** in which we process the courses.

Step-by-Step Logic:

- Build a graph and in-degree array as before.
- Start with courses with 0 prerequisites.
- Each time you process a course, add it to the result list.
- If you process all courses, return the result. Otherwise, return an empty list (cycle detected).

Pseudocode:

```
function findOrder(numCourses, prerequisites):
    create graph as adjacency list
    create in_degree array of size numCourses
    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()
        add course to result
        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 Input/Output:

- Input: `numCourses = 4`, `prerequisites = [[1,0],[2,0],[3,1],[3,2]]`
- Output: `[0,1,2,3]` or `[0,2,1,3]`
- Both are valid: 0 first, then 1/2, then 3.

Another Test Case for You:

- Input: `numCourses = 2`, `prerequisites = [[0,1],[1,0]]`
- Output: `[]` (Cycle)

Step-by-step Trace for Example:

- Build in_degree: [0,1,1,2]
- queue: [0]
- Pop 0: result=[0], add 1,2 to queue (both in_degree become 0)
- Pop 1: result=[0,1], decrease in_degree[3] to 1
- Pop 2: result=[0,1,2], decrease in_degree[3] to 0, add 3 to queue
- Pop 3: result=[0,1,2,3]

Time and Space Complexity:

- Both O(N + E) (same as before).

Problem 3: Parallel Courses ([LeetCode Link](https://leetcode.com/problems/parallel-courses/))

What's new here?

You're still given `n` courses and a list of prerequisite pairs, but now you can take **any number of courses per semester** as long as their prerequisites are done. The goal: **Find the minimum number of semesters to finish all courses**.

Why is this harder?

Instead of finding an order, you need to "batch" courses together, taking as many as possible each semester.

Brute Force:

Try all ways to schedule courses in parallel batches—exponential, not feasible.

Optimal Approach: BFS by Levels (Layered Topological Sort)

- Each "layer" (courses with zero in-degree at the same time) is a semester.
- At each semester, take all available courses (with zero prerequisites).
- Remove these from the graph, repeat for the next semester.

Pseudocode:

```
function minimumSemesters(n, relations):
    build graph and in_degree array
    queue = all courses with in_degree 0
    semester = 0
    courses_taken = 0

    while queue not empty:
        next_queue = []
        for course in queue:
            courses_taken += 1
            for neighbor in graph[course]:
                in_degree[neighbor] -= 1
                if in_degree[neighbor] == 0:
                    next_queue.append(neighbor)
        queue = next_queue
        semester += 1

    if courses_taken == n:
        return semester
    else:
        return -1  # Cycle detected
```

Example Input/Output:

- Input: `n = 3`, `relations = [[1,3],[2,3]]`
- Output: `2`
- Explanation: Take [1,2] in semester 1, [3] in semester 2.

Another Test Case:

- Input: `n = 4`, `relations = [[2,1],[3,1],[1,4]]`
- Output: `3`
- Try to batch the courses:

- Semester 1: [2,3]
- Semester 2: [1]
- Semester 3: [4]

Time and Space Complexity:

- O(N + E), for building the graph and level-order traversal.

Summary and Next Steps

Recap:

You just navigated three classic dependency resolution problems—all with topological sort as the core pattern.

- **Course Schedule:** Can you finish? (Cycle detection)
- **Course Schedule II:** Give a valid order. (Ordering)
- **Parallel Courses:** What’s the minimum number of batches/semesters? (Layered BFS)

Key Patterns/Insights:

- Use in-degree arrays to track prerequisites.
- BFS is great for both ordering and batching.
- If you process fewer nodes than exist, there's a cycle.
- For "minimum semesters", process by BFS levels.

Common Mistakes:

- Not resetting/initializing in-degree or graph correctly.
- Forgetting to check for cycles.
- Confusing edge directions (src vs dest).
- Not returning early on cycle detection.

Action List

- Try implementing all three problems yourself—even the one with code provided!
- For Problems 2 and 3, see if you can solve them with DFS instead of BFS.
- Search for other topological sort or dependency problems—practice makes perfect.
- Compare your code with solutions from others, especially for edge cases.
- If you get stuck, pause and draw the dependency graph by hand—it helps!

You’re building a powerful graph intuition—keep going!