Topic Introduction

Today we're diving into the world of **graph ordering and optimal tree roots**—a family of problems where the order and structure of nodes matter. The main concept tying these together is the **Topological Sort**, a cornerstone of many graph-based interview questions.

What is Topological Sort?
Topological sorting is an ordering of 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. Imagine you're building a house: you must pour the foundation before framing the walls, and install plumbing before drywall. Topological sort gives you the valid order to perform such dependent tasks.

How does it work?
There are two main ways:

- **Kahn's Algorithm** (BFS): Repeatedly remove nodes with zero incoming edges (in-degree 0), and decrease the in-degree of their neighbors.
- **DFS-based:** Visit nodes recursively, putting them on a stack after visiting all their dependencies.

When is it useful?

- Scheduling tasks with dependencies
- Resolving build orders (like in software projects)
- Understanding order constraints (like learning an alien dictionary!)

Example (not one of today's problems):
Suppose you have tasks: `A` depends on `B` and `C`, `C` depends on `B`.
Valid order: `B, C, A` (since B before C, C before A).

Today's featured problems:

- Alien Dictionary: Find the letter order in an unknown language from a list of sorted words.
- Sequence Reconstruction: Validate if a unique topological order exists from given subsequences.
- Minimum Height Trees: Find the root nodes for minimum height trees in an undirected graph.

Why are these grouped together?
All three ask you to reason about the order or arrangement of nodes based on connections. The first two are classic topological sort questions, while the third, though undirected, asks you to think about optimal tree structures from a "rooting" perspective—a different flavor of the same core idea.

Problem 1: Alien Dictionary

Leetcode Link

Problem Statement (Rephrased):
Given a list of words sorted lexicographically in an unknown language, figure out the order of the letters. If it's impossible, return an empty string.

Example Input/Output:

Input: `["wrt","wrf","er","ett","rftt"]`

Output: `"wertf"`

How do you think about it?

Imagine the words are sorted as per the alien language. Each time a letter in one word comes before a different letter in the next word, you learn their relative order. Your goal: build a directed graph of letter dependencies, and topologically sort it.

Try drawing the graph by hand for a small set of words. It helps!

Try this test case:

Input: `["z","x","z"]`

What should the output be?

Brute-force idea:

Try all possible letter permutations and check if they satisfy the order for every adjacent pair of words.

Time: O(N! * M) (N = num letters, M = total letters in all words). Not practical.

Optimal approach:

- Build a graph: for each pair of adjacent words, find the first differing character. That gives you an edge.
- Topologically sort the graph (using Kahn's algorithm or DFS).
- If you detect a cycle or can't order all letters, return "".

Let's code it:

```python
from collections import defaultdict, deque

def alienOrder(words):
    # Step 1: Build graph and in-degree count
    graph = defaultdict(set)
    in_degree = {c: 0 for word in words for c in word}  # all unique letters

    # Step 2: Add edges
    for i in range(len(words) - 1):
        w1, w2 = words[i], words[i + 1]
        min_len = min(len(w1), len(w2))
        # Check the first difference
        for j in range(min_len):
            if w1[j] != w2[j]:
                if w2[j] not in graph[w1[j]]:
                    graph[w1[j]].add(w2[j])
                    in_degree[w2[j]] += 1
                break
        else:
            # Check for invalid case: prefix order
            if len(w1) > len(w2):
                return ""
    # Step 3: Topological sort (BFS)
    queue = deque([c for c in in_degree if in_degree[c] == 0])
    order = []
    while queue:
        c = queue.popleft()
        order.append(c)
        for nei in graph[c]:
            in_degree[nei] -= 1
            if in_degree[nei] == 0:
                queue.append(nei)

    # Step 4: Check if all letters are in result
    if len(order) == len(in_degree):
        return "".join(order)
    else:
        return ""
```

Complexity:

- Time: O(N + A) (N = total letters, A = unique letters)
- Space: O(A^2) (for graph and in-degree)

Code walkthrough:

- We first collect all unique letters as graph nodes.
- For each adjacent word pair, we add a directed edge based on the first character that differs.
- If there’s a word that is a prefix of another but comes after, we return "".
- We use a queue to process all nodes with in-degree 0 (no dependencies).
- If we process all letters, we return the order; else, there’s a cycle or disconnected part.

Trace Example:

Input: `["wrt","wrf","er","ett","rftt"]`

- Compare "wrt" and "wrf" -> 't' before 'f'
- "wrf" and "er" -> 'w' before 'e'
- "er" and "ett" -> 'r' before 't'
- "ett" and "rftt" -> 'e' before 'r'

Graph:

w: [e]

e: [r]

r: [t]

t: [f]

f: []

In-degree:

w:0, e:1, r:1, t:1, f:1

Queue starts with 'w'.

Order: w e r t f

Try this yourself:

Input: `["abc","ab"]`

What should the output be?

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

Problem 2: Sequence Reconstruction

Leetcode Link

Problem Statement (Rephrased):

Given an original sequence and a list of subsequences, can you uniquely reconstruct the original sequence using these subsequences? (A unique topological order must exist.)

How is this related?

Just like Alien Dictionary, we're inferring an order from partial information. But here, you check if the ordering is unique and matches a target.

Brute-force idea:

List all possible topological sorts and check if only one matches the target.

Way too slow (there can be exponential sorts).

Optimal approach:

- Build a graph from the subsequences.
- Topologically sort, but at each step, ensure only one node is available (i.e., in-degree 0). If at any time more than one node is available, the order is not unique.
- The reconstructed order must match the original.

Step-by-step:

- Construct a graph and in-degree table from all numbers in `seqs`.
- Use Kahn's algorithm:
  - At every step, if there are multiple nodes with in-degree 0, return False.
  - Otherwise, remove the node and update neighbors.
- At the end, the sequence you constructed must be identical to `org`.

Pseudocode:

```
function sequenceReconstruction(org, seqs):
    Build graph and in-degree for all unique numbers in seqs
    If org and set of nodes differ, return False

    queue = all nodes with in-degree 0
    idx = 0

    while queue is not empty:
        If len(queue) > 1: return False
        node = queue.pop()
        If idx >= len(org) or org[idx] != node: return False
        idx += 1
        For each neighbor:
            reduce in-degree
            If in-degree == 0, add to queue

    return idx == len(org)
```

Example:

org = `[1,2,3]`, seqs = `[[1,2],[1,3]]`

Graph: 1->2, 1->3

Orderings possible: [1,2,3] and [1,3,2] (not unique)

Output: False

Try this:

org = `[4,1,5,2,6,3]`, seqs = `[[5,2,6,3],[4,1,5,2]]`

What should the output be?

Test case walkthrough:

org = `[1,2,3]`, seqs = `[[1,2],[2,3]]`

Edges: 1->2, 2->3

Only possible order: 1,2,3

At each step, only one node with in-degree 0

Output: True

Complexity:

- Time: O(N + E) (N = nodes, E = edges)
- Space: O(N + E)

Problem 3: Minimum Height Trees

Leetcode Link

Problem Statement (Rephrased):

Given an undirected graph (tree) with n nodes, find all root nodes that would result in a minimum height tree.

What's different here?

This is not a classic topological sort, but it uses similar intuition: peeling off leaves level by level to find the center(s), which are the optimal roots.

Brute-force:

Try rooting the tree at every node, compute the height each time.

Way too slow: O(n^2).

Optimal approach:

- Iteratively remove all leaves (nodes with 1 neighbor), like peeling an onion.

- The last one or two nodes left are the centers (roots of minimum height trees).

Pseudocode:

```
function findMinHeightTrees(n, edges):
    If n == 1: return [0]
    Build adjacency list
    Find all leaves (degree 1 nodes)
    While n > 2:
        n -= number of leaves
        Remove current leaves and update neighbors
        Add new leaves to the list
    Return remaining nodes
```

Example:

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

Step-by-step:

- Initial leaves: [0,1,2,5]
- Remove them: [3,4] remain (centers)
- Output: [3,4]

Try this:

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

What is the output?

Complexity:

- Time: O(n)
- Space: O(n)

Hint:

Notice how this process is a "reverse" topological sort, cutting leaves rather than building order from roots.

Summary and Next Steps

Today, you explored three problems united by the concept of **graph ordering and rooting**:

- **Topological sort** is a must-know for interview graph questions about order and dependencies.
- Sometimes, the task is to find a unique ordering (as in Sequence Reconstruction).
- Other times, you need to find optimal centers in undirected trees (Minimum Height Trees), which is like root-finding from the opposite direction.

Key patterns to remember:

- Build your graph and in-degree table carefully.
- Know how to detect cycles and ambiguous orderings.
- For undirected trees, iterative leaf removal is a powerful trick.

Common mistakes:

- Not handling prefix edge cases in Alien Dictionary.
- Not checking for multiple zero in-degree nodes (ambiguity) in Sequence Reconstruction.
- Forgetting that a tree's center can be **two nodes** if the tree's height is even.

Action List

- Try solving all three problems on your own, even if code was provided.
- For Problem 2 and 3, see if you can implement both BFS (queue) and DFS (recursive) solutions.
- Explore other problems that use topological sort or tree rooting, like Course Schedule or Tree Diameter.
- Compare your solutions with others, especially on edge cases (like cycles or disconnected graphs).
- If you get stuck, don't worry! Break problems down, draw graphs, and keep practicing.

Happy coding—today, you're one step closer to being a graph order ninja!