LeetCode Training Day 16 Topology Sort

Topology sort is a common problem in graph explorer. Generally speaking we have a number of nodes we need to visited, however, due to the dependencies between nodes, we have to visit all its dependent nodes before visiting the node itself. This generates the problem of topology sort.

For Topology sort, we will use an array to record the dependencies for each node, we call it as degree, which comes from how many incoming edges to the node. We can visit any nodes with degree as zero, but we can not visit any node with degree more than zero.

After we visit a node, we need to decrease the degree for all the nodes which depend on this node and when we see a new node with degree as zero we will add it to the process queue.

The core data structure for toplogy sort is an array of degree for each node, an hashtable mapping to a list of nodes to see which nodes depend on current node and a process queue to process all nodes with degree zero. The process sequence is not relevant.

The core code pattern is a BFS for graph traverse.

## 207. Course Schedule

Medium

There are a total of numCourses courses you have to take, labeled from 0 to numCourses - 1. You are given an array prerequisites where prerequisites[i] = [ai, bi] indicates that you **must** take course bi first if you want to take course ai.

* For example, the pair [0, 1], indicates that to take course 0 you have to first take course 1.

Return true if you can finish all courses. Otherwise, return false.

**Example 1:**

**Input:** numCourses = 2, prerequisites = [[1,0]]

**Output:** true

**Explanation:** There are a total of 2 courses to take.

To take course 1 you should have finished course 0. So it is possible.

**Example 2:**

**Input:** numCourses = 2, prerequisites = [[1,0],[0,1]]

**Output:** false

**Explanation:** There are a total of 2 courses to take.

To take course 1 you should have finished course 0, and to take course 0 you should also have finished course 1. So it is impossible.

**Constraints:**

* 1 <= numCourses <= 105
* 0 <= prerequisites.length <= 5000
* prerequisites[i].length == 2
* 0 <= ai, bi < numCourses
* All the pairs prerequisites[i] are **unique**.

### Analysis:

For any prerequistite, we add one degree to the course, and we collect the course with degree as zero. In the end we compare the courses we took and all the courses.

/// <summary>

/// LeetCode #207. Course Schedule

/// There are a total of n courses you have to take, labeled from 0 to

/// n - 1.

/// Some courses may have prerequisites, for example to take course 0 you

/// have to first take course 1, which is expressed as a pair: [0,1]

/// Given the total number of courses and a list of prerequisite pairs,

/// is it possible for you to finish all courses?

///

/// For example:

/// 2, [[1,0]]

/// There are a total of 2 courses to take. To take course 1 you should

/// have finished course 0. So it is possible.

///

/// 2, [[1,0],[0,1]]

/// There are a total of 2 courses to take. To take course 1 you should

/// have finished course 0,

/// and to take course 0 you should also have finished course 1. So it

/// is impossible.

/// Note:

/// The input prerequisites is a graph represented by a list of edges,

/// not adjacency matrices. Read more about how a graph is represented.

/// </summary>

bool LeetCodeGraph::canFinishCourse(int numCourses, vector<vector<int>>& prerequisites)

{

vector<int> degree(numCourses);

vector<unordered\_set<int>> dependency(numCourses);

queue<int> search;

// remember which course dependes on others and which ones depends on me

for (size\_t i = 0; i < prerequisites.size(); i++)

{

degree[prerequisites[i][0]]++;

dependency[prerequisites[i][1]].insert(prerequisites[i][0]);

}

int result = 0;

// get all the course not depends on others, this is our starting search scope

for (size\_t i = 0; i < degree.size(); i++)

{

if (degree[i] == 0)

{

search.push(i);

result++;

}

}

// Using queue to manage BFS and get every free course and clear the

// dependency with a free course, i.e. you depend on a free course,

// then such dependency

// does not matter. If all dependencies are clear, we got a new

// free course

while (!search.empty())

{

int free\_course = search.front();

search.pop();

for (int next\_course : dependency[free\_course])

{

degree[next\_course]--;

if (degree[next\_course] == 0)

{

search.push(next\_course);

result++;

}

}

}

// if number of free courses equals to the total course, we can finish

// all courses

return result == numCourses;

}

## 210. Course Schedule II

Medium

There are a total of numCourses courses you have to take, labeled from 0 to numCourses - 1. You are given an array prerequisites where prerequisites[i] = [ai, bi] indicates that you **must** take course bi first if you want to take course ai.

* For example, the pair [0, 1], indicates that to take course 0 you have to first take course 1.

Return *the ordering of courses you should take to finish all courses*. If there are many valid answers, return **any** of them. If it is impossible to finish all courses, return **an empty array**.

**Example 1:**

**Input:** numCourses = 2, prerequisites = [[1,0]]

**Output:** [0,1]

**Explanation:** There are a total of 2 courses to take. To take course 1 you should have finished course 0. So the correct course order is [0,1].

**Example 2:**

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

**Output:** [0,2,1,3]

**Explanation:** There are a total of 4 courses to take. To take course 3 you should have finished both courses 1 and 2. Both courses 1 and 2 should be taken after you finished course 0.

So one correct course order is [0,1,2,3]. Another correct ordering is [0,2,1,3].

**Example 3:**

**Input:** numCourses = 1, prerequisites = []

**Output:** [0]

**Constraints:**

* 1 <= numCourses <= 2000
* 0 <= prerequisites.length <= numCourses \* (numCourses - 1)
* prerequisites[i].length == 2
* 0 <= ai, bi < numCourses
* ai != bi
* All the pairs [ai, bi] are **distinct**.

### Analysis:

For any prerequistite, we add one degree to the course, and we collect the course with degree as zero. In the end we compare the courses we took and all the courses.

/// <summary>

/// LeetCode #210. Course Schedule II

///

/// There are a total of n courses you have to take, labeled from 0 to

/// n - 1.

/// Some courses may have prerequisites, for example to take course 0 you

/// have to first take course 1, which is expressed as a pair: [0,1]

/// Given the total number of courses and a list of prerequisite pairs,

/// return the ordering of courses you should take to finish all courses.

///

/// There may be multiple correct orders, you just need to return one of

/// them. If it is impossible to finish all courses, return an empty array.

/// For example:

/// 2, [[1,0]]

///

/// There are a total of 2 courses to take. To take course 1 you should

/// have finished course 0. So the correct course order is [0,1]

/// 4, [[1,0],[2,0],[3,1],[3,2]]

/// There are a total of 4 courses to take. To take course 3 you should

/// have finished both courses 1 and 2.

/// Both courses 1 and 2 should be taken after you finished course 0.

/// So one correct course order is [0,1,2,3].

/// Another correct ordering is[0,2,1,3].

/// Note:

/// The input prerequisites is a graph represented by a list of edges,

/// not adjacency matrices. Read more about how a graph is represented.

/// click to show more hints.

/// Hints:

/// 1.This problem is equivalent to finding the topological order in a

/// directed graph.

/// If a cycle exists, no topological ordering exists and therefore

//// it will be impossible to take all courses.

/// 2.Topological Sort via DFS - A great video tutorial (21 minutes) on

/// Coursera explaining the basic concepts of Topological Sort.

/// 3.Topological sort could also be done via BFS.

/// </summary>

vector<int> LeetCodeGraph::findOrder(int numCourses, vector<vector<int>>& prerequisites)

{

vector<int> degree(numCourses);

vector<unordered\_set<int>> dependency(numCourses);

queue<int> search;

// remember which course dependes on others and which ones depends on me

for (size\_t i = 0; i < prerequisites.size(); i++)

{

degree[prerequisites[i][0]]++;

dependency[prerequisites[i][1]].insert(prerequisites[i][0]);

}

// get all the course not depends on others, this is our starting search scope

vector<int> result;

for (size\_t i = 0; i < degree.size(); i++)

{

if (degree[i] == 0)

{

search.push(i);

result.push\_back(i);

}

}

while (!search.empty())

{

int free\_course = search.front();

search.pop();

for (int next\_course : dependency[free\_course])

{

degree[next\_course]--;

if (degree[next\_course] == 0)

{

search.push(next\_course);

result.push\_back(next\_course);

}

}

}

if (result.size() == numCourses)

{

return result;

}

else

{

return vector<int>();

}

}

## 261. Graph Valid Tree

Medium

You have a graph of n nodes labeled from 0 to n - 1. You are given an integer n and a list of edges where edges[i] = [ai, bi] indicates that there is an undirected edge between nodes ai and bi in the graph.

Return true *if the edges of the given graph make up a valid tree, and* false *otherwise*.

**Example 1:**

Diagram, schematic

Description automatically generated

**Input:** n = 5, edges = [[0,1],[0,2],[0,3],[1,4]]

**Output:** true

**Example 2:**

A picture containing text, clock, clipart

Description automatically generated

**Input:** n = 5, edges = [[0,1],[1,2],[2,3],[1,3],[1,4]]

**Output:** false

**Constraints:**

* 1 <= n <= 2000
* 0 <= edges.length <= 5000
* edges[i].length == 2
* 0 <= ai, bi < n
* ai != bi
* There are no self-loops or repeated edges.

### Analysis:

Because the graph is represented as undirected edge, you have to build dual direction. For any nodes which are leaves should have only one connection.

/// <summary>

/// Leet code #261. Graph Valid Tree

///

/// Given n nodes labeled from 0 to n - 1 and a list of undirected edges

/// (each edge is a pair of nodes), write a function to check whether these

/// edges make up a valid tree.

/// For example:

/// Given n = 5 and edges = [[0, 1], [0, 2], [0, 3], [1, 4]], return true.

/// Given n = 5 and edges = [[0, 1], [1, 2], [2, 3], [1, 3], [1, 4]],

/// return false.

/// Hint:

/// 1.Given n = 5 and edges = [[0, 1], [1, 2], [3, 4]], what should your

/// return?

/// Is this case a valid tree?

/// 2.According to the definition of tree on Wikipedia: "a tree is an

/// undirected graph in which any two vertices are connected by exactly

/// one path. In other words, any connected graph without simple

/// cycles is a tree."

/// </summary>

bool LeetCodeGraph::validTree(int n, vector<pair<int, int>>& edges)

{

unordered\_map<int, unordered\_set<int>> tree\_map;

queue<int> process\_queue;

for (size\_t i = 0; i < edges.size(); i++)

{

tree\_map[edges[i].first].insert(edges[i].second);

tree\_map[edges[i].second].insert(edges[i].first);

}

for (auto itr : tree\_map)

{

if (itr.second.size() == 1)

{

process\_queue.push(itr.first);

}

}

int count = 0;

while (!process\_queue.empty())

{

int node = process\_queue.front();

process\_queue.pop();

if (tree\_map[node].size() == 0) continue;

int parent = \*tree\_map[node].begin();

tree\_map[parent].erase(node);

tree\_map.erase(node);

if (tree\_map[parent].size() == 1)

{

process\_queue.push(parent);

}

count++;

}

if (count == n - 1) return true;

else return false;

}

## 310. Minimum Height Trees

Medium

A tree is an undirected graph in which any two vertices are connected by *exactly* one path. In other words, any connected graph without simple cycles is a tree.

Given a tree of n nodes labelled from 0 to n - 1, and an array of n - 1 edges where edges[i] = [ai, bi] indicates that there is an undirected edge between the two nodes ai and bi in the tree, you can choose any node of the tree as the root. When you select a node x as the root, the result tree has height h. Among all possible rooted trees, those with minimum height (i.e. min(h))  are called **minimum height trees** (MHTs).

Return *a list of all****MHTs'****root labels*. You can return the answer in **any order**.

The **height** of a rooted tree is the number of edges on the longest downward path between the root and a leaf.

**Example 1:**

A picture containing text, clock, clipart

Description automatically generated

**Input:** n = 4, edges = [[1,0],[1,2],[1,3]]

**Output:** [1]

**Explanation:** As shown, the height of the tree is 1 when the root is the node with label 1 which is the only MHT.

**Example 2:**

A picture containing clock

Description automatically generated

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

**Output:** [3,4]

**Constraints:**

* 1 <= n <= 2 \* 104
* edges.length == n - 1
* 0 <= ai, bi < n
* ai != bi
* All the pairs (ai, bi) are distinct.
* The given input is **guaranteed** to be a tree and there will be **no repeated** edges.

### Analysis:

We use neighbors to set dual direction edges. We start nodes with only one connection as leaves and cut leaves.

/// <summary>

/// Leet code #310. Minimum Height Trees

///

/// For a undirected graph with tree characteristics, we can choose any node as the root.

/// The result graph is then a rooted tree. Among all possible rooted trees, those with

/// minimum height are called minimum height trees (MHTs).

/// Given such a graph, write a function to find all the MHTs and return a list of their root labels.

///

/// Format

/// The graph contains n nodes which are labeled from 0 to n - 1. You will be given

/// the number n and a list of undirected edges (each edge is a pair of labels).

///

/// You can assume that no duplicate edges will appear in edges. Since all edges are

/// undirected, [0, 1] is the same as [1, 0] and thus will not appear together in edges.

///

/// Example 1:

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

/// 0

/// |

/// 1

/// / \

/// 2 3

///

/// return [1]

///

/// Example 2:

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

/// 0 1 2

/// \ | /

/// 3

/// |

/// 4

/// |

/// 5

///

/// return [3, 4]

/// Hint:

/// 1.How many MHTs can a graph have at most?

/// Note:

///

/// (1) According to the definition of tree on Wikipedia: “a tree is an undirected graph in

/// which any two vertices are connected by exactly one path. In other words, any connected

/// graph without simple cycles is a tree.”

/// (2) The height of a rooted tree is the number of edges on the longest downward path between the root and a leaf.

/// </summary>

vector<int> LeetCodeGraph::findMinHeightTrees(int n, vector<pair<int, int>>& edges)

{

vector<unordered\_set<int>> route\_set(n);

vector<int> result;

for (size\_t i = 0; i < edges.size(); i++)

{

route\_set[edges[i].first].insert(edges[i].second);

route\_set[edges[i].second].insert(edges[i].first);

}

set<int> leaves, new\_leaves;

for (int i = 0; i < n; i++)

{

if (route\_set[i].size() <= 1) leaves.insert(i);

}

while (true)

{

n = n - leaves.size();

if ((n == 0) || leaves.empty()) break;

for (int leaf : leaves)

{

int target = \*route\_set[leaf].begin();

route\_set[leaf].erase(target);

route\_set[target].erase(leaf);

if (route\_set[target].size() <= 1) new\_leaves.insert(target);

}

leaves = new\_leaves;

new\_leaves.clear();

}

for (int leaf : leaves)

{

result.push\_back(leaf);

}

return result;

}

## 582. Kill Process

Medium

You have n processes forming a rooted tree structure. You are given two integer arrays pid and ppid, where pid[i] is the ID of the ith process and ppid[i] is the ID of the ith process's parent process.

Each process has only **one parent process** but may have multiple children processes. Only one process has ppid[i] = 0, which means this process has **no parent process** (the root of the tree).

When a process is **killed**, all of its children processes will also be killed.

Given an integer kill representing the ID of a process you want to kill, return *a list of the IDs of the processes that will be killed. You may return the answer in****any order****.*

**Example 1:**

A picture containing text, clipart

Description automatically generated

**Input:** pid = [1,3,10,5], ppid = [3,0,5,3], kill = 5

**Output:** [5,10]

**Explanation:** The processes colored in red are the processes that should be killed.

**Example 2:**

**Input:** pid = [1], ppid = [0], kill = 1

**Output:** [1]

**Constraints:**

* n == pid.length
* n == ppid.length
* 1 <= n <= 5 \* 104
* 1 <= pid[i] <= 5 \* 104
* 0 <= ppid[i] <= 5 \* 104
* Only one process has no parent.
* All the values of pid are **unique**.
* kill is **guaranteed** to be in pid.

### Analysis:

Very typical topology sort. The parent processes are dependencies, we do not need degree because we know the starting process.

/// <summary>

/// Leet code #582. Kill Process

///

/// Given n processes, each process has a unique PID (process id) and its

/// PPID (parent process id).

/// Each process only has one parent process, but may have one or more

/// children processes. This is just like a tree structure. Only one

/// process has PPID that is 0, which means this process has no parent

/// process. All the PIDs will be distinct positive integers.

///

/// We use two list of integers to represent a list of processes, where

/// the first list contains PID for each process and the second list

/// contains the corresponding PPID.

///

/// Now given the two lists, and a PID representing a process you want

/// to kill, return a list of PIDs of processes that will be killed in

/// the end. You should assume that when a process is killed,

/// all its children processes will be killed.

/// No order is required for the final answer.

/// Example 1:

/// Input:

/// pid = [1, 3, 10, 5]

/// ppid = [3, 0, 5, 3]

/// kill = 5

/// Output: [5,10]

/// Explanation:

/// 3

/// / \

/// 1 5

/// /

/// 10

/// Kill 5 will also kill 10.

/// Note:

/// 1. The given kill id is guaranteed to be one of the given PIDs.

/// 2. n >= 1.

/// </summary>

vector<int> LeetCodeGraph::killProcess(vector<int>& pid, vector<int>& ppid, int kill)

{

unordered\_map<int, set<int>> process\_map;

queue<int> process\_queue;

vector<int> result;

for (size\_t i = 0; i < pid.size(); i++)

{

process\_map[ppid[i]].insert(pid[i]);

}

process\_queue.push(kill);

while (!process\_queue.empty())

{

int process\_id = process\_queue.front();

process\_queue.pop();

result.push\_back(process\_id);

for (int p : process\_map[process\_id])

{

process\_queue.push(p);

}

}

return result;

}

## 802. Find Eventual Safe States

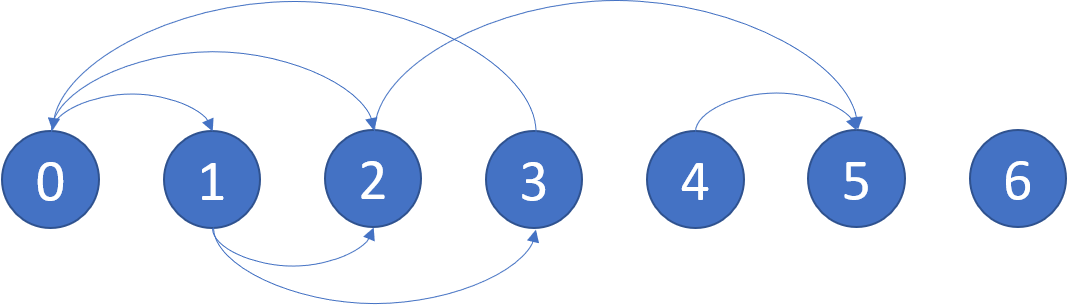
Medium

There is a directed graph of n nodes with each node labeled from 0 to n - 1. The graph is represented by a **0-indexed** 2D integer array graph where graph[i] is an integer array of nodes adjacent to node i, meaning there is an edge from node i to each node in graph[i].

A node is a **terminal node** if there are no outgoing edges. A node is a **safe node** if every possible path starting from that node leads to a **terminal node**.

Return *an array containing all the****safe nodes****of the graph*. The answer should be sorted in **ascending** order.

**Example 1:**



**Input:** graph = [[1,2],[2,3],[5],[0],[5],[],[]]

**Output:** [2,4,5,6]

**Explanation:** The given graph is shown above.

Nodes 5 and 6 are terminal nodes as there are no outgoing edges from either of them.

Every path starting at nodes 2, 4, 5, and 6 all lead to either node 5 or 6.

**Example 2:**

**Input:** graph = [[1,2,3,4],[1,2],[3,4],[0,4],[]]

**Output:** [4]

**Explanation:**

Only node 4 is a terminal node, and every path starting at node 4 leads to node 4.

**Constraints:**

* n == graph.length
* 1 <= n <= 104
* 0 <= graph[i].length <= n
* 0 <= graph[i][j] <= n - 1
* graph[i] is sorted in a strictly increasing order.
* The graph may contain self-loops.
* The number of edges in the graph will be in the range [1, 4 \* 104].

### Analysis:

Starting from terminal nodes, reversely do topology sort.

/// <summary>

/// Leet code #802. Find Eventual Safe States

///

/// In a directed graph, we start at some node and every turn, walk along

/// a directed edge of the graph. If we reach a node that is terminal

/// (that is, it has no outgoing directed edges), we stop.

///

/// Now, say our starting node is eventually safe if and only if we must

/// eventually walk to a terminal node. More specifically, there exists a

/// natural number K so that for any choice of where to walk, we must have

/// stopped at a terminal node in less than K steps.

///

/// Which nodes are eventually safe? Return them as an array in sorted

/// order.

///

/// The directed graph has N nodes with labels 0, 1, ..., N-1, where N is

/// the length of graph. The graph is given in the following form:

/// graph[i] is a list of labels j such that (i, j) is a directed edge of

/// the graph.

///

/// Example:

/// Input: graph = [[1,2],[2,3],[5],[0],[5],[],[]]

/// Output: [2,4,5,6]

/// Here is a diagram of the above graph.

///

/// Illustration of graph

///

/// Note:

///

/// 1. graph will have length at most 10000.

/// 3. The number of edges in the graph will not exceed 32000.

/// 3. Each graph[i] will be a sorted list of different integers, chosen

/// within the range [0, graph.length - 1].

/// </summary>

vector<int> LeetCodeGraph::eventualSafeNodes(vector<vector<int>>& graph)

{

unordered\_map<int, unordered\_set<int>> next\_map;

unordered\_map<int, unordered\_set<int>> prev\_map;

vector<int> result;

queue<int> search;

for (size\_t i = 0; i < graph.size(); i++)

{

for (size\_t j = 0; j < graph[i].size(); j++)

{

next\_map[i].insert(graph[i][j]);

prev\_map[graph[i][j]].insert(i);

}

if (graph[i].empty()) search.push(i);

}

while (!search.empty())

{

int front = search.front();

search.pop();

result.push\_back(front);

for (int prev : prev\_map[front])

{

next\_map[prev].erase(front);

if (next\_map[prev].empty()) search.push(prev);

}

prev\_map.erase(front);

}

sort(result.begin(), result.end());

return result;

}

## 851. Loud and Rich

Medium

There is a group of n people labeled from 0 to n - 1 where each person has a different amount of money and a different level of quietness.

You are given an array richer where richer[i] = [ai, bi] indicates that ai has more money than bi and an integer array quiet where quiet[i] is the quietness of the ith person. All the given data in richer are **logically correct** (i.e., the data will not lead you to a situation where x is richer than y and y is richer than x at the same time).

Return *an integer array*answer*where*answer[x] = y*if*y*is the least quiet person (that is, the person*y*with the smallest value of*quiet[y]*) among all people who definitely have equal to or more money than the person*x.

**Example 1:**

**Input:** richer = [[1,0],[2,1],[3,1],[3,7],[4,3],[5,3],[6,3]], quiet = [3,2,5,4,6,1,7,0]

**Output:** [5,5,2,5,4,5,6,7]

**Explanation:**

answer[0] = 5.

Person 5 has more money than 3, which has more money than 1, which has more money than 0.

The only person who is quieter (has lower quiet[x]) is person 7, but it is not clear if they have more money than person 0.

answer[7] = 7.

Among all people that definitely have equal to or more money than person 7 (which could be persons 3, 4, 5, 6, or 7), the person who is the quietest (has lower quiet[x]) is person 7.

The other answers can be filled out with similar reasoning.

**Example 2:**

**Input:** richer = [], quiet = [0]

**Output:** [0]

**Constraints:**

* n == quiet.length
* 1 <= n <= 500
* 0 <= quiet[i] < n
* All the values of quiet are **unique**.
* 0 <= richer.length <= n \* (n - 1) / 2
* 0 <= ai, bi < n
* ai!= bi
* All the pairs of richer are **unique**.
* The observations in richer are all logically consistent.

### Analysis:

By default, every element in result pointing to themselves. Then starting from the richest person to the poorest using topology sort, replace the elements by the louder person in iteration.

/// <summary>

/// Leet code #851. Loud and Rich

///

/// In a group of N people (labelled 0, 1, 2, ..., N-1), each person has

/// different amounts of money, and different levels of quietness.

///

/// For convenience, we'll call the person with label x, simply "person x".

///

/// We'll say that richer[i] = [x, y] if person x definitely has more money

/// than person y. Note that richer may only be a subset of valid

/// observations.

///

/// Also, we'll say quiet[x] = q if person x has quietness q.

///

/// Now, return answer, where answer[x] = y if y is the least quiet person

/// (that is, the person y with the smallest value of quiet[y]), among all

/// people who definitely have equal to or more money than person x.

///

/// Example 1:

///

/// Input: richer = [[1,0],[2,1],[3,1],[3,7],[4,3],[5,3],[6,3]],

/// quiet = [3,2,5,4,6,1,7,0]

/// Output: [5,5,2,5,4,5,6,7]

/// Explanation:

/// answer[0] = 5.

/// Person 5 has more money than 3, which has more money than 1, which has

/// more money than 0.

/// The only person who is quieter (has lower quiet[x]) is person 7, but

/// it isn't clear if they have more money than person 0.

///

/// answer[7] = 7.

/// Among all people that definitely have equal to or more money than person 7

/// (which could be persons 3, 4, 5, 6, or 7), the person who is the quietest

/// (has lower quiet[x]) is person 7.

///

/// The other answers can be filled out with similar reasoning.

/// Note:

///

/// 1. 1 <= quiet.length = N <= 500

/// 2. 0 <= quiet[i] < N, all quiet[i] are different.

/// 3. 0 <= richer.length <= N \* (N-1) / 2

/// 4. 0 <= richer[i][j] < N

/// 5. richer[i][0] != richer[i][1]

/// 6. richer[i]'s are all different.

/// 7. The observations in richer are all logically consistent.

/// </summary>

vector<int> LeetCodeGraph::loudAndRich(vector<vector<int>>& richer, vector<int>& quiet)

{

vector<int> result(quiet.size());

for (size\_t i = 0; i < result.size(); i++) result[i] = i;

vector<unordered\_set<int>> relation\_map(quiet.size());

vector<int> count\_map(quiet.size());

for (size\_t i = 0; i < richer.size(); i++)

{

relation\_map[richer[i][0]].insert(richer[i][1]);

count\_map[richer[i][1]]++;

}

queue<int> search;

for (size\_t i = 0; i < count\_map.size(); i++)

{

if (count\_map[i] == 0) search.push(i);

}

while (!search.empty())

{

int person = search.front();

search.pop();

if (quiet[person] < quiet[result[person]])

{

result[person] = person;

}

for (auto next : relation\_map[person])

{

if (quiet[result[person]] < quiet[result[next]])

{

result[next] = result[person];

}

count\_map[next]--;

if (count\_map[next] == 0) search.push(next);

}

}

return result;

}

## 1136. Parallel Courses

Medium

You are given an integer n, which indicates that there are n courses labeled from 1 to n. You are also given an array relations where relations[i] = [prevCoursei, nextCoursei], representing a prerequisite relationship between course prevCoursei and course nextCoursei: course prevCoursei has to be taken before course nextCoursei.

In one semester, you can take **any number** of courses as long as you have taken all the prerequisites in the **previous** semester for the courses you are taking.

Return *the****minimum****number of semesters needed to take all courses*. If there is no way to take all the courses, return -1.

**Example 1:**

Diagram

Description automatically generated

**Input:** n = 3, relations = [[1,3],[2,3]]

**Output:** 2

**Explanation:** The figure above represents the given graph.

In the first semester, you can take courses 1 and 2.

In the second semester, you can take course 3.

**Example 2:**

Diagram

Description automatically generated

**Input:** n = 3, relations = [[1,2],[2,3],[3,1]]

**Output:** -1

**Explanation:** No course can be studied because they are prerequisites of each other.

**Constraints:**

* 1 <= n <= 5000
* 1 <= relations.length <= 5000
* relations[i].length == 2
* 1 <= prevCoursei, nextCoursei <= n
* prevCoursei != nextCoursei
* All the pairs [prevCoursei, nextCoursei] are **unique**.

### Analysis:

Use topology sort, run the courses in parallel by layers.

/// <summary>

/// Leet code #1136. Parallel Courses

///

/// There are N courses, labelled from 1 to N.

///

/// We are given relations[i] = [X, Y], representing a prerequisite

/// relationship between course X and course Y: course X has to be

/// studied before course Y.

///

/// In one semester you can study any number of courses as long as

/// you have studied all the prerequisites for the course you are

/// studying.

///

/// Return the minimum number of semesters needed to study all

/// courses. If there is no way to study all the courses, return -1.

///

/// Example 1:

/// Input: N = 3, relations = [[1,3],[2,3]]

/// Output: 2

/// Explanation:

/// In the first semester, courses 1 and 2 are studied. In the second

/// semester, course 3 is studied.

///

/// Example 2:

/// Input: N = 3, relations = [[1,2],[2,3],[3,1]]

/// Output: -1

/// Explanation:

/// No course can be studied because they depend on each other.

///

/// Note:

/// 1. 1 <= N <= 5000

/// 2. 1 <= relations.length <= 5000

/// 3. relations[i][0] != relations[i][1]

/// 4. There are no repeated relations in the input.

/// </summary>

int LeetCodeGraph::minimumSemesters(int N, vector<vector<int>>& relations)

{

vector<int> degree(N);

vector<vector<int>> dependency(N);

for (size\_t i = 0; i < relations.size(); i++)

{

int x = relations[i][0] - 1;

int y = relations[i][1] - 1;

degree[y]++;

dependency[x].push\_back(y);

}

queue<int> search;

int result = 0;

for (size\_t i = 0; i < degree.size(); i++)

{

if (degree[i] == 0) search.push(i);

}

int count = 0;

while (!search.empty())

{

size\_t size = search.size();

for (size\_t i = 0; i < size; i++)

{

int course = search.front();

search.pop();

count++;

for (size\_t j = 0; j < dependency[course].size(); j++)

{

int next = dependency[course][j];

degree[next]--;

if (degree[next] == 0) search.push(next);

}

}

result++;

}

if (count == N) return result;

else return -1;

}

## 1245. Tree Diameter

Medium

Given an undirected tree, return its diameter: the number of **edges** in a longest path in that tree.

The tree is given as an array of edges where edges[i] = [u, v] is a bidirectional edge between nodes u and v.  Each node has labels in the set {0, 1, ..., edges.length}.

**Example 1:**

Diagram, shape, arrow

Description automatically generated

**Input:** edges = [[0,1],[0,2]]

**Output:** 2

**Explanation:**

A longest path of the tree is the path 1 - 0 - 2.

**Example 2:**

A picture containing watch

Description automatically generated

**Input:** edges = [[0,1],[1,2],[2,3],[1,4],[4,5]]

**Output:** 4

**Explanation:**

A longest path of the tree is the path 3 - 2 - 1 - 4 - 5.

**Constraints:**

* 0 <= edges.length < 10^4
* edges[i][0] != edges[i][1]
* 0 <= edges[i][j] <= edges.length
* The given edges form an undirected tree.

### Analysis:

Cut from leaves to root, every time add diameters by 2.

/// <summary>

/// Leet code #1245. Tree Diameter

///

/// Given an undirected tree, return its diameter: the number of edges in a

/// longest path in that tree.

///

/// The tree is given as an array of edges where edges[i] = [u, v] is a

/// bidirectional edge between nodes u and v. Each node has labels in the

/// set {0, 1, ..., edges.length}.

///

/// Example 1:

///

/// Input: edges = [[0,1],[0,2]]

/// Output: 2

/// Explanation:

/// A longest path of the tree is the path 1 - 0 - 2.

///

/// Example 2:

///

/// Input: edges = [[0,1],[1,2],[2,3],[1,4],[4,5]]

/// Output: 4

/// Explanation:

/// A longest path of the tree is the path 3 - 2 - 1 - 4 - 5.

///

/// Constraints:

/// 1. 0 <= edges.length < 10^4

/// 2. edges[i][0] != edges[i][1]

/// 3. 0 <= edges[i][j] <= edges.length

/// 4. The given edges form an undirected tree.

/// </summary>

int LeetCodeGraph::treeDiameter(vector<vector<int>>& edges)

{

unordered\_map<int, set<int>> node\_map;

for (size\_t i = 0; i < edges.size(); i++)

{

node\_map[edges[i][0]].insert(edges[i][1]);

node\_map[edges[i][1]].insert(edges[i][0]);

}

queue<int> search;

for (auto itr : node\_map)

{

if (itr.second.size() == 1)

{

search.push(itr.first);

}

}

int result = 0;

int count = node\_map.size();

while (!search.empty())

{

size\_t size = search.size();

if (count > 2) result += 2;

else if (count == 2) result += 1;

for (size\_t i = 0; i < size; i++)

{

int node = search.front();

search.pop();

count--;

for (auto target : node\_map[node])

{

node\_map[target].erase(node);

if (node\_map[target].size() == 1)

{

search.push(target);

}

}

}

}

return result;

}

## 1273. Delete Tree Nodes

Medium

A tree rooted at node 0 is given as follows:

* The number of nodes is nodes;
* The value of the ith node is value[i];
* The parent of the ith node is parent[i].

Remove every subtree whose sum of values of nodes is zero.

Return *the number of the remaining nodes in the tree*.

**Example 1:**

A picture containing schematic

Description automatically generated

**Input:** nodes = 7, parent = [-1,0,0,1,2,2,2], value = [1,-2,4,0,-2,-1,-1]

**Output:** 2

**Example 2:**

**Input:** nodes = 7, parent = [-1,0,0,1,2,2,2], value = [1,-2,4,0,-2,-1,-2]

**Output:** 6

**Constraints:**

* 1 <= nodes <= 104
* parent.length == nodes
* 0 <= parent[i] <= nodes - 1
* parent[0] == -1 which indicates that 0 is the root.
* value.length == nodes
* -105 <= value[i] <= 105
* The given input is **guaranteed** to represent a **valid tree**.

### Analysis:

Use topology search, starting from leaves (the nodes with no one else pointing to them as parent).

/// <summary>

/// Leet code #1273. Delete Tree Nodes

///

/// A tree rooted at node 0 is given as follows:

///

/// The number of nodes is nodes;

/// The value of the i-th node is value[i];

/// The parent of the i-th node is parent[i].

/// Remove every subtree whose sum of values of nodes is zero.

///

/// After doing so, return the number of nodes remaining in the tree.

///

/// Example 1:

///

/// Input: nodes = 7, parent = [-1,0,0,1,2,2,2],

/// value = [1,-2,4,0,-2,-1,-1]

///

/// Output: 2

///

/// Constraints:

/// 1. 1 <= nodes <= 10^4

/// 2. -10^5 <= value[i] <= 10^5

/// 3. parent.length == nodes

/// 4. parent[0] == -1 which indicates that 0 is the root.

/// </summary>

int LeetCodeGraph::deleteTreeNodes(int nodes, vector<int>& parent, vector<int>& value)

{

vector<int> degree(nodes);

vector<int> sum(nodes), count(nodes);

for (size\_t i = 1; i < parent.size(); i++)

{

degree[parent[i]]++;

}

queue<int> search;

for (size\_t i = 0; i < degree.size(); i++)

{

if (degree[i] == 0) search.push(i);

}

while (!search.empty())

{

int node = search.front();

search.pop();

sum[node] += value[node];

if (sum[node] != 0) count[node]++;

else count[node] = 0;

int parent\_node = parent[node];

if (parent\_node == -1) continue;

sum[parent\_node] += sum[node];

count[parent\_node] += count[node];

degree[parent\_node]--;

if (degree[parent\_node] == 0)

{

search.push(parent\_node);

}

}

return count[0];

}

## 1466. Reorder Routes to Make All Paths Lead to the City Zero

Medium

There are n cities numbered from 0 to n - 1 and n - 1 roads such that there is only one way to travel between two different cities (this network form a tree). Last year, The ministry of transport decided to orient the roads in one direction because they are too narrow.

Roads are represented by connections where connections[i] = [ai, bi] represents a road from city ai to city bi.

This year, there will be a big event in the capital (city 0), and many people want to travel to this city.

Your task consists of reorienting some roads such that each city can visit the city 0. Return the **minimum** number of edges changed.

It's **guaranteed** that each city can reach city 0 after reorder.

**Example 1:**

Diagram

Description automatically generated

**Input:** n = 6, connections = [[0,1],[1,3],[2,3],[4,0],[4,5]]

**Output:** 3

**Explanation:** Change the direction of edges show in red such that each node can reach the node 0 (capital).

**Example 2:**

A picture containing table

Description automatically generated

**Input:** n = 5, connections = [[1,0],[1,2],[3,2],[3,4]]

**Output:** 2

**Explanation:** Change the direction of edges show in red such that each node can reach the node 0 (capital).

**Example 3:**

**Input:** n = 3, connections = [[1,0],[2,0]]

**Output:** 0

**Constraints:**

* 2 <= n <= 5 \* 104
* connections.length == n - 1
* connections[i].length == 2
* 0 <= ai, bi <= n - 1
* ai != bi

### Analysis:

Use topology search, but since the edge has directions we use matrix not neighbors to represent the routes.

/// <summary>

/// Leet code #1466. Reorder Routes to Make All Paths Lead to the City Zero

///

/// Medium

///

/// There are n cities numbered from 0 to n-1 and n-1 roads such that

/// there is only one way to travel between two different cities (this

/// network form a tree). Last year, The ministry of transport decided to

/// orient the roads in one direction because they are too narrow.

///

/// Roads are represented by connections where connections[i] = [a, b]

/// represents a road from city a to b.

///

/// This year, there will be a big event in the capital (city 0), and

/// many people want to travel to this city.

///

/// Your task consists of reorienting some roads such that each city

/// can visit the city 0. Return the minimum number of edges changed.

///

/// It's guaranteed that each city can reach the city 0 after reorder.

///

/// Example 1:

/// Input: n = 6, connections = [[0,1],[1,3],[2,3],[4,0],[4,5]]

/// Output: 3

/// Explanation: Change the direction of edges show in red such that

/// each node can reach the node 0 (capital).

///

/// Example 2:

/// Input: n = 5, connections = [[1,0],[1,2],[3,2],[3,4]]

/// Output: 2

/// Explanation: Change the direction of edges show in red such that

/// each node can reach the node 0 (capital).

///

/// Example 3:

/// Input: n = 3, connections = [[1,0],[2,0]]

/// Output: 0

///

/// Constraints:

/// 1. 2 <= n <= 5 \* 10^4

/// 2. connections.length == n-1

/// 3. connections[i].length == 2

/// 4. 0 <= connections[i][0], connections[i][1] <= n-1

/// 5. connections[i][0] != connections[i][1]

/// </summary>

int LeetCodeGraph::minReorder(int n, vector<vector<int>>& connections)

{

vector<unordered\_map<int, int>> road(n);

for (size\_t i = 0; i < connections.size(); i++)

{

road[connections[i][0]][connections[i][1]] = 1;

road[connections[i][1]][connections[i][0]] = -1;

}

queue<int> search;

int result = 0;

search.push(0);

while (!search.empty())

{

int city = search.front();

search.pop();

for (auto itr : road[city])

{

if (itr.second == 0) continue;

if (itr.second == 1) result++;

road[city][itr.first] = 0;

road[itr.first][city] = 0;

search.push(itr.first);

}

}

return result;

}

## 1557. Minimum Number of Vertices to Reach All Nodes

Medium

Given a**directed acyclic graph**, with n vertices numbered from 0 to n-1, and an array edges where edges[i] = [fromi, toi] represents a directed edge from node fromi to node toi.

Find *the smallest set of vertices from which all nodes in the graph are reachable*. It's guaranteed that a unique solution exists.

Notice that you can return the vertices in any order.

**Example 1:**

Application, background pattern

Description automatically generated

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

**Output:** [0,3]

**Explanation:** It's not possible to reach all the nodes from a single vertex. From 0 we can reach [0,1,2,5]. From 3 we can reach [3,4,2,5]. So we output [0,3].

**Example 2:**

Application

Description automatically generated

**Input:** n = 5, edges = [[0,1],[2,1],[3,1],[1,4],[2,4]]

**Output:** [0,2,3]

**Explanation:** Notice that vertices 0, 3 and 2 are not reachable from any other node, so we must include them. Also any of these vertices can reach nodes 1 and 4.

**Constraints:**

* 2 <= n <= 10^5
* 1 <= edges.length <= min(10^5, n \* (n - 1) / 2)
* edges[i].length == 2
* 0 <= fromi, toi < n
* All pairs (fromi, toi) are distinct.

### Analysis:

We only need the nodes with no dependencies, (degree as zero)

/// <summary>

/// Leet code #1557. Minimum Number of Vertices to Reach All Nodes

///

/// Medium

///

/// Given a directed acyclic graph, with n vertices numbered from 0 to

/// n-1, and an array edges where edges[i] = [fromi, toi] represents

/// a directed edge from node fromi to node toi.

/// Find the smallest set of vertices from which all nodes in the graph

/// are reachable. It's guaranteed that a unique solution exists.

///

/// Notice that you can return the vertices in any order.

///

/// Example 1:

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

/// Output: [0,3]

/// Explanation: It's not possible to reach all the nodes from a

/// single vertex. From 0 we can reach [0,1,2,5]. From 3 we can

/// reach [3,4,2,5]. So we output [0,3].

///

/// Example 2:

/// Input: n = 5, edges = [[0,1],[2,1],[3,1],[1,4],[2,4]]

/// Output: [0,2,3]

/// Explanation: Notice that vertices 0, 3 and 2 are not reachable from

/// any other node, so we must include them. Also any of these vertices

/// can reach nodes 1 and 4.

///

/// Constraints:

/// 1. 2 <= n <= 10^5

/// 2. 1 <= edges.length <= min(10^5, n \* (n - 1) / 2)

/// 3. edges[i].length == 2

/// 3. 0 <= fromi, toi < n

/// 4. All pairs (fromi, toi) are distinct.

/// </summary>

vector<int> LeetCodeGraph::findSmallestSetOfVertices(int n, vector<vector<int>>& edges)

{

vector<int> degree(n);

vector<int> result;

for (size\_t i = 0; i < edges.size(); i++)

{

degree[edges[i][1]]++;

}

for (size\_t i = 0; i < degree.size(); i++)

{

if (degree[i] == 0)

{

result.push\_back(i);

}

}

return result;

}

## 1743. Restore the Array From Adjacent Pairs

Medium

There is an integer array nums that consists of n **unique**elements, but you have forgotten it. However, you do remember every pair of adjacent elements in nums.

You are given a 2D integer array adjacentPairs of size n - 1 where each adjacentPairs[i] = [ui, vi] indicates that the elements ui and vi are adjacent in nums.

It is guaranteed that every adjacent pair of elements nums[i] and nums[i+1] will exist in adjacentPairs, either as [nums[i], nums[i+1]] or [nums[i+1], nums[i]]. The pairs can appear **in any order**.

Return *the original array*nums*. If there are multiple solutions, return****any of them***.

**Example 1:**

**Input:** adjacentPairs = [[2,1],[3,4],[3,2]]

**Output:** [1,2,3,4]

**Explanation:** This array has all its adjacent pairs in adjacentPairs.

Notice that adjacentPairs[i] may not be in left-to-right order.

**Example 2:**

**Input:** adjacentPairs = [[4,-2],[1,4],[-3,1]]

**Output:** [-2,4,1,-3]

**Explanation:** There can be negative numbers.

Another solution is [-3,1,4,-2], which would also be accepted.

**Example 3:**

**Input:** adjacentPairs = [[100000,-100000]]

**Output:** [100000,-100000]

**Constraints:**

* nums.length == n
* adjacentPairs.length == n - 1
* adjacentPairs[i].length == 2
* 2 <= n <= 105
* -105 <= nums[i], ui, vi <= 105
* There exists some nums that has adjacentPairs as its pairs.

### Analysis:

Use topology sort, starting with number never appears as second node in pair.

/// <summary>

/// Leet code 1743. Restore the Array From Adjacent Pairs

///

/// Medium

///

/// There is an integer array nums that consists of n unique elements,

/// but you have forgotten it. However, you do remember every pair of

/// adjacent elements in nums.

///

/// You are given a 2D integer array adjacentPairs of size n - 1 where

/// each adjacentPairs[i] = [ui, vi] indicates that the elements ui and

/// vi are adjacent in nums.

///

/// It is guaranteed that every adjacent pair of elements nums[i] and

/// nums[i+1] will exist in adjacentPairs, either as [nums[i],

/// nums[i+1]] or [nums[i+1], nums[i]]. The pairs can appear in any order.

///

/// Return the original array nums. If there are multiple solutions,

/// return any of them.

///

/// Example 1:

/// Input: adjacentPairs = [[2,1],[3,4],[3,2]]

/// Output: [1,2,3,4]

/// Explanation: This array has all its adjacent pairs in adjacentPairs.

/// Notice that adjacentPairs[i] may not be in left-to-right order.

///

/// Example 2:

/// Input: adjacentPairs = [[4,-2],[1,4],[-3,1]]

/// Output: [-2,4,1,-3]

/// Explanation: There can be negative numbers.

/// Another solution is [-3,1,4,-2], which would also be accepted.

///

/// Example 3:

/// Input: adjacentPairs = [[100000,-100000]]

/// Output: [100000,-100000]

///

/// Constraints:

/// 1. nums.length == n

/// 2. adjacentPairs.length == n - 1

/// 3. adjacentPairs[i].length == 2

/// 4. 2 <= n <= 10^5

/// 5. -10^5 <= nums[i], ui, vi <= 10^5

/// 6. There exists some nums that has adjacentPairs as its pairs.

/// </summary>

vector<int> LeetCodeGraph::restoreArray(vector<vector<int>>& adjacentPairs)

{

unordered\_set<int> head;

unordered\_map<int, unordered\_set<int>> adjacent;

for (size\_t i = 0; i < adjacentPairs.size(); i++)

{

int a = adjacentPairs[i][0];

int b = adjacentPairs[i][1];

if (head.count(a) > 0) head.erase(a);

else head.insert(a);

if (head.count(b) > 0) head.erase(b);

else head.insert(b);

adjacent[a].insert(b);

adjacent[b].insert(a);

}

vector<int> result;

int start = \*head.begin();

result.push\_back(start);

while (!adjacent.empty())

{

int next = \*adjacent[start].begin();

adjacent[next].erase(start);

if (adjacent[next].empty()) adjacent.erase(next);

adjacent.erase(start);

start = next;

result.push\_back(start);

}

return result;

}

## 2049. Count Nodes With the Highest Score

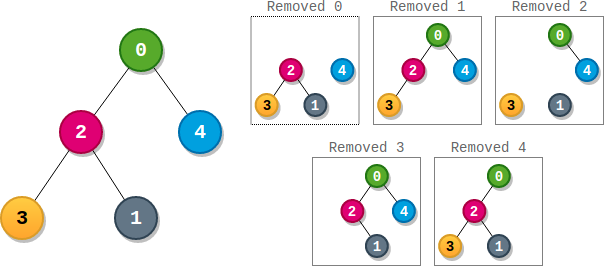
Medium

There is a **binary** tree rooted at 0 consisting of n nodes. The nodes are labeled from 0 to n - 1. You are given a **0-indexed** integer array parents representing the tree, where parents[i] is the parent of node i. Since node 0 is the root, parents[0] == -1.

Each node has a **score**. To find the score of a node, consider if the node and the edges connected to it were **removed**. The tree would become one or more **non-empty** subtrees. The **size** of a subtree is the number of the nodes in it. The **score** of the node is the **product of the sizes** of all those subtrees.

Return *the****number****of nodes that have the****highest score***.

**Example 1:**



**Input:** parents = [-1,2,0,2,0]

**Output:** 3

**Explanation:**

- The score of node 0 is: 3 \* 1 = 3

- The score of node 1 is: 4 = 4

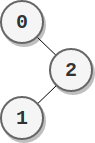
- The score of node 2 is: 1 \* 1 \* 2 = 2

- The score of node 3 is: 4 = 4

- The score of node 4 is: 4 = 4

The highest score is 4, and three nodes (node 1, node 3, and node 4) have the highest score.

**Example 2:**



**Input:** parents = [-1,2,0]

**Output:** 2

**Explanation:**

- The score of node 0 is: 2 = 2

- The score of node 1 is: 2 = 2

- The score of node 2 is: 1 \* 1 = 1

The highest score is 2, and two nodes (node 0 and node 1) have the highest score.

**Constraints:**

* n == parents.length
* 2 <= n <= 105
* parents[0] == -1
* 0 <= parents[i] <= n - 1 for i != 0
* parents represents a valid binary tree.

### Analysis:

We use topology sort traverse from leaves, on each node, count two children (if available), and others in tree (total size - children - 1).

/// <summary>

/// Leet Code 2049. Count Nodes With the Highest Score

///

/// Medium

///

/// There is a binary tree rooted at 0 consisting of n nodes. The nodes

/// are labeled from 0 to n - 1. You are given a 0-indexed integer

/// array parents representing the tree, where parents[i] is

/// the parent of node i. Since node 0 is the root, parents[0] == -1.

///

/// Each node has a score. To find the score of a node, consider if

/// the node and the edges connected to it were removed. The tree would

/// become one or more non-empty subtrees. The size of a subtree is the

/// number of the nodes in it. The score of the node is the product of

/// the sizes of all those subtrees.

///

/// Return the number of nodes that have the highest score.

///

/// Example 1:

/// Input: parents = [-1,2,0,2,0]

/// Output: 3

/// Explanation:

/// - The score of node 0 is: 3 \* 1 = 3

/// - The score of node 1 is: 4 = 4

/// - The score of node 2 is: 1 \* 1 \* 2 = 2

/// - The score of node 3 is: 4 = 4

/// - The score of node 4 is: 4 = 4

/// The highest score is 4, and three nodes (node 1, node 3, and node 4)

/// have the highest score.

///

/// Example 2:

/// Input: parents = [-1,2,0]

/// Output: 2

/// Explanation:

/// - The score of node 0 is: 2 = 2

/// - The score of node 1 is: 2 = 2

/// - The score of node 2 is: 1 \* 1 = 1

/// The highest score is 2, and two nodes (node 0 and node 1) have the

/// highest score.

///

/// Constraints:

/// 1. n == parents.length

/// 2. 2 <= n <= 10^5

/// 3. parents[0] == -1

/// 4. 0 <= parents[i] <= n - 1 for i != 0

/// 5. parents represents a valid binary tree.

/// </summary>

int LeetCodeGraph::countHighestScoreNodes(vector<int>& parents)

{

vector<int> degree(parents.size());

vector<long long> score(parents.size(), 1);

vector<int> count(parents.size(), 1);

int result = 0;

long long max\_score = 0;

for (size\_t i = 1; i < parents.size(); i++)

{

degree[parents[i]]++;

}

queue<int> queue;

for (size\_t i = 0; i < degree.size(); i++)

{

if (degree[i] == 0) queue.push(i);

}

while (!queue.empty())

{

int node = queue.front();

queue.pop();

if(node != 0) score[node] = ((long long)parents.size() - (long long)count[node]) \* score[node];

if (score[node] > max\_score)

{

result = 1;

max\_score = score[node];

}

else if (score[node] == max\_score)

{

result++;

}

if (node == 0) break;

int parent = parents[node];

score[parent] = score[parent] \* (long long)count[node];

count[parent] += count[node];

degree[parent]--;

if (degree[parent] == 0)

{

queue.push(parent);

}

}

return result;

}

## 2115. Find All Possible Recipes from Given Supplies

Medium

You have information about n different recipes. You are given a string array recipes and a 2D string array ingredients. The ith recipe has the name recipes[i], and you can **create** it if you have **all** the needed ingredients from ingredients[i]. Ingredients to a recipe may need to be created from **other**recipes, i.e., ingredients[i] may contain a string that is in recipes.

You are also given a string array supplies containing all the ingredients that you initially have, and you have an infinite supply of all of them.

Return *a list of all the recipes that you can create.*You may return the answer in **any order**.

Note that two recipes may contain each other in their ingredients.

**Example 1:**

**Input:** recipes = ["bread"], ingredients = [["yeast","flour"]], supplies = ["yeast","flour","corn"]

**Output:** ["bread"]

**Explanation:**

We can create "bread" since we have the ingredients "yeast" and "flour".

**Example 2:**

**Input:** recipes = ["bread","sandwich"], ingredients = [["yeast","flour"],["bread","meat"]], supplies = ["yeast","flour","meat"]

**Output:** ["bread","sandwich"]

**Explanation:**

We can create "bread" since we have the ingredients "yeast" and "flour".

We can create "sandwich" since we have the ingredient "meat" and can create the ingredient "bread".

**Example 3:**

**Input:** recipes = ["bread","sandwich","burger"], ingredients = [["yeast","flour"],["bread","meat"],["sandwich","meat","bread"]], supplies = ["yeast","flour","meat"]

**Output:** ["bread","sandwich","burger"]

**Explanation:**

We can create "bread" since we have the ingredients "yeast" and "flour".

We can create "sandwich" since we have the ingredient "meat" and can create the ingredient "bread".

We can create "burger" since we have the ingredient "meat" and can create the ingredients "bread" and "sandwich".

**Constraints:**

* n == recipes.length == ingredients.length
* 1 <= n <= 100
* 1 <= ingredients[i].length, supplies.length <= 100
* 1 <= recipes[i].length, ingredients[i][j].length, supplies[k].length <= 10
* recipes[i], ingredients[i][j], and supplies[k] consist only of lowercase English letters.
* All the values of recipes and supplies combined are unique.
* Each ingredients[i] does not contain any duplicate values.

### Analysis:

We start from supplies, make the recipes depending on all ingredients, if all ingredients are ready the recipe is ready.

/// <summary>

/// Leet Code 2115. Find All Possible Recipes from Given Supplies

///

/// Medium

///

/// You have information about n different recipes. You are given a string

/// array recipes and a 2D string array ingredients. The ith recipe has

/// the name recipes[i], and you can create it if you have all the needed

/// ingredients from ingredients[i]. Ingredients to a recipe may need to

/// be created from other recipes, i.e., ingredients[i] may contain a

/// string that is in recipes.

///

/// You are also given a string array supplies containing all the

/// ingredients that you initially have, and you have an infinite supply

/// of all of them.

///

/// Return a list of all the recipes that you can create. You may return

/// the answer in any order.

///

/// Note that two recipes may contain each other in their ingredients.

///

/// Example 1:

/// Input: recipes = ["bread"], ingredients = [["yeast","flour"]],

/// supplies = ["yeast","flour","corn"]

/// Output: ["bread"]

/// Explanation:

/// We can create "bread" since we have the ingredients "yeast" and

/// "flour".

///

/// Example 2:

/// Input: recipes = ["bread","sandwich"], ingredients = [["yeast",

/// "flour"],["bread","meat"]], supplies = ["yeast","flour","meat"]

/// Output: ["bread","sandwich"]

/// Explanation:

/// We can create "bread" since we have the ingredients "yeast" and

/// "flour".

/// We can create "sandwich" since we have the ingredient "meat" and

/// can create the ingredient "bread".

///

/// Example 3:

/// Input: recipes = ["bread","sandwich","burger"], ingredients =

/// [["yeast","flour"],["bread","meat"],["sandwich","meat","bread"]],

/// supplies = ["yeast","flour","meat"]

/// Output: ["bread","sandwich","burger"]

///

/// Explanation:

/// We can create "bread" since we have the ingredients "yeast" and

/// "flour".

/// We can create "sandwich" since we have the ingredient "meat" and

/// can create the ingredient "bread".

/// We can create "burger" since we have the ingredient "meat" and

/// can create the ingredients "bread" and "sandwich".

///

/// Constraints:

/// 1. n == recipes.length == ingredients.length

/// 2. 1 <= n <= 100

/// 3. 1 <= ingredients[i].length, supplies.length <= 100

/// 4. 1 <= recipes[i].length, ingredients[i][j].length,

/// supplies[k].length <= 10

/// 5. recipes[i], ingredients[i][j], and supplies[k] consist only

/// of lowercase English letters.

/// 6. All the values of recipes and supplies combined are unique.

/// 7. Each ingredients[i] does not contain any duplicate values.

/// </summary>

vector<string> LeetCodeGraph::findAllRecipes(vector<string>& recipes,

vector<vector<string>>& ingredients,

vector<string>& supplies)

{

vector<int> degree(recipes.size());

unordered\_map<string, vector<int>> dependencies;

for (size\_t i = 0; i < ingredients.size(); i++)

{

degree[i] = ingredients[i].size();

for (size\_t j = 0; j < ingredients[i].size(); j++)

{

dependencies[ingredients[i][j]].push\_back(i);

}

}

queue<string> queue;

for (size\_t i = 0; i < supplies.size(); i++)

{

queue.push(supplies[i]);

}

vector<string> result;

while (!queue.empty())

{

string ingredient = queue.front();

queue.pop();

for (size\_t i = 0; i < dependencies[ingredient].size(); i++)

{

int recipe\_id = dependencies[ingredient][i];

degree[recipe\_id]--;

if (degree[recipe\_id] == 0)

{

queue.push(recipes[recipe\_id]);

result.push\_back(recipes[recipe\_id]);

}

}

}

return result;

}

## 444. Sequence Reconstruction

Medium

You are given an integer array nums of length n where nums is a permutation of the integers in the range [1, n]. You are also given a 2D integer array sequences where sequences[i] is a subsequence of nums.

Check if nums is the shortest possible and the only **supersequence**. The shortest **supersequence** is a sequence **with the shortest length** and has all sequences[i] as subsequences. There could be multiple valid **supersequences** for the given array sequences.

* For example, for sequences = [[1,2],[1,3]], there are two shortest **supersequences**, [1,2,3] and [1,3,2].
* While for sequences = [[1,2],[1,3],[1,2,3]], the only shortest **supersequence** possible is [1,2,3]. [1,2,3,4] is a possible supersequence but not the shortest.

Return true*if*nums*is the only shortest****supersequence****for*sequences*, or*false*otherwise*.

A **subsequence** is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements.

**Example 1:**

**Input:** nums = [1,2,3], sequences = [[1,2],[1,3]]

**Output:** false

**Explanation:** There are two possible supersequences: [1,2,3] and [1,3,2].

The sequence [1,2] is a subsequence of both: [**1**,**2**,3] and [**1**,3,**2**].

The sequence [1,3] is a subsequence of both: [**1**,2,**3**] and [**1**,**3**,2].

Since nums is not the only shortest supersequence, we return false.

**Example 2:**

**Input:** nums = [1,2,3], sequences = [[1,2]]

**Output:** false

**Explanation:** The shortest possible supersequence is [1,2].

The sequence [1,2] is a subsequence of it: [**1**,**2**].

Since nums is not the shortest supersequence, we return false.

**Example 3:**

**Input:** nums = [1,2,3], sequences = [[1,2],[1,3],[2,3]]

**Output:** true

**Explanation:** The shortest possible supersequence is [1,2,3].

The sequence [1,2] is a subsequence of it: [**1**,**2**,3].

The sequence [1,3] is a subsequence of it: [**1**,2,**3**].

The sequence [2,3] is a subsequence of it: [1,**2**,**3**].

Since nums is the only shortest supersequence, we return true.

**Constraints:**

* n == nums.length
* 1 <= n <= 104
* nums is a permutation of all the integers in the range [1, n].
* 1 <= sequences.length <= 104
* 1 <= sequences[i].length <= 104
* 1 <= sum(sequences[i].length) <= 105
* 1 <= sequences[i][j] <= n
* All the arrays of sequences are **unique**.
* sequences[i] is a subsequence of nums.

### Analysis:

We run topology sort based on sequences, on every turn we should only see one free node, i.e. the queue size should be 1 only, because if you see both x and y are free, then you can not determine x and y which comes first. You also need to check the sequence match the target. Please notice the sequence can become circle or incomplete, you need to check final result size against target nums size.

/// <summary>

/// Leet Code 444. Sequence Reconstruction

///

/// Medium

///

/// You are given an integer array nums of length n where nums is a

/// permutation of the integers in the range [1, n]. You are also

/// given a 2D integer array sequences where sequences[i] is a

/// subsequence of nums.

///

/// Check if nums is the shortest possible and the only supersequence.

/// The shortest supersequence is a sequence with the shortest length

/// and has all sequences[i] as subsequences. There could be multiple

/// valid supersequences for the given array sequences.

///

/// For example, for sequences = [[1,2],[1,3]], there are two shortest

/// supersequences, [1,2,3] and [1,3,2].

/// While for sequences = [[1,2],[1,3],[1,2,3]], the only shortest

/// supersequence possible is [1,2,3]. [1,2,3,4] is a possible

/// supersequence but not the shortest.

/// Return true if nums is the only shortest supersequence for

/// sequences, or false otherwise.

///

/// A subsequence is a sequence that can be derived from another sequence

/// by deleting some or no elements without changing the order of the

/// remaining elements.

///

/// Example 1:

/// Input: nums = [1,2,3], sequences = [[1,2],[1,3]]

/// Output: false

/// Explanation: There are two possible supersequences: [1,2,3] and

/// [1,3,2].

/// The sequence [1,2] is a subsequence of both: [1,2,3] and [1,3,2].

/// The sequence [1,3] is a subsequence of both: [1,2,3] and [1,3,2].

/// Since nums is not the only shortest supersequence, we return false.

///

/// Example 2:

/// Input: nums = [1,2,3], sequences = [[1,2]]

/// Output: false

/// Explanation: The shortest possible supersequence is [1,2].

/// The sequence [1,2] is a subsequence of it: [1,2].

/// Since nums is not the shortest supersequence, we return false.

///

/// Example 3:

/// Input: nums = [1,2,3], sequences = [[1,2],[1,3],[2,3]]

/// Output: true

/// Explanation: The shortest possible supersequence is [1,2,3].

/// The sequence [1,2] is a subsequence of it: [1,2,3].

/// The sequence [1,3] is a subsequence of it: [1,2,3].

/// The sequence [2,3] is a subsequence of it: [1,2,3].

/// Since nums is the only shortest supersequence, we return true.

///

/// Constraints:

/// 1. n == nums.length

/// 2. 1 <= n <= 10^4

/// 3. nums is a permutation of all the integers in the range [1, n].

/// 4. 1 <= sequences.length <= 10^4

/// 5. 1 <= sequences[i].length <= 10^4

/// 6. 1 <= sum(sequences[i].length) <= 10^5

/// 7. 1 <= sequences[i][j] <= n

/// 8. All the arrays of sequences are unique.

/// 9. sequences[i] is a subsequence of nums.

/// </summary>

bool LeetCodeGraph::sequenceReconstruction(vector<int>& nums, vector<vector<int>>& sequences)

{

vector<int> degree(nums.size() + 1);

vector<vector<int>> dependencies(nums.size() + 1);

for (size\_t i = 0; i < sequences.size(); i++)

{

for (int j = sequences[i].size() - 1; j > 0; j--)

{

if (sequences[i][j] > (int)nums.size()) return false;

degree[sequences[i][j]]++;

dependencies[sequences[i][j - 1]].push\_back(sequences[i][j]);

}

}

queue<int> queue;

vector<int> result;

for (size\_t i = 1; i < degree.size(); i++)

{

if (degree[i] == 0) queue.push(i);

}

while (!queue.empty())

{

if (queue.size() > 1) return false;

int node = queue.front();

queue.pop();

if (node != nums[result.size()]) return false;

result.push\_back(node);

for (size\_t i = 0; i < dependencies[node].size(); i++)

{

degree[dependencies[node][i]]--;

if (degree[dependencies[node][i]] == 0)

{

queue.push(dependencies[node][i]);

}

}

}

if (result.size() != nums.size()) return false;

else return true;

}

# Advance Problems

## 2050. Parallel Courses III

Hard

You are given an integer n, which indicates that there are n courses labeled from 1 to n. You are also given a 2D integer array relations where relations[j] = [prevCoursej, nextCoursej] denotes that course prevCoursej has to be completed **before** course nextCoursej (prerequisite relationship). Furthermore, you are given a **0-indexed** integer array time where time[i] denotes how many **months** it takes to complete the (i+1)th course.

You must find the **minimum** number of months needed to complete all the courses following these rules:

* You may start taking a course at **any time** if the prerequisites are met.
* **Any number of courses** can be taken at the **same time**.

Return *the****minimum****number of months needed to complete all the courses*.

**Note:** The test cases are generated such that it is possible to complete every course (i.e., the graph is a directed acyclic graph).

**Example 1:**

**A picture containing text, pool ball, sport, pool table

Description automatically generated**

**Input:** n = 3, relations = [[1,3],[2,3]], time = [3,2,5]

**Output:** 8

**Explanation:** The figure above represents the given graph and the time required to complete each course.

We start course 1 and course 2 simultaneously at month 0.

Course 1 takes 3 months and course 2 takes 2 months to complete respectively.

Thus, the earliest time we can start course 3 is at month 3, and the total time required is 3 + 5 = 8 months.

**Example 2:**

**A picture containing text, pool ball, vector graphics

Description automatically generated**

**Input:** n = 5, relations = [[1,5],[2,5],[3,5],[3,4],[4,5]], time = [1,2,3,4,5]

**Output:** 12

**Explanation:** The figure above represents the given graph and the time required to complete each course.

You can start courses 1, 2, and 3 at month 0.

You can complete them after 1, 2, and 3 months respectively.

Course 4 can be taken only after course 3 is completed, i.e., after 3 months. It is completed after 3 + 4 = 7 months.

Course 5 can be taken only after courses 1, 2, 3, and 4 have been completed, i.e., after max(1,2,3,7) = 7 months.

Thus, the minimum time needed to complete all the courses is 7 + 5 = 12 months.

**Constraints:**

* 1 <= n <= 5 \* 104
* 0 <= relations.length <= min(n \* (n - 1) / 2, 5 \* 104)
* relations[j].length == 2
* 1 <= prevCoursej, nextCoursej <= n
* prevCoursej != nextCoursej
* All the pairs [prevCoursej, nextCoursej] are **unique**.
* time.length == n
* 1 <= time[i] <= 104
* The given graph is a directed acyclic graph.

### Analysis:

We use topology sort to process the courses, on every course, when we process its dependencies, we also calculate the start time for this course as the latest finish time of its dependencies.

/// <summary>

/// Leet Code 2050. Parallel Courses III

///

/// Hard

///

/// You are given an integer n, which indicates that there are n courses

/// labeled from 1 to n. You are also given a 2D integer array relations

/// where relations[j] = [prevCoursej, nextCoursej] denotes that course

/// prevCoursej has to be completed before course nextCoursej

/// (prerequisite relationship). Furthermore, you are given a 0-indexed

/// integer array time where time[i] denotes how many months it takes

/// to complete the (i+1)th course.

///

/// You must find the minimum number of months needed to complete all

/// the courses following these rules:

///

/// You may start taking a course at any time if the prerequisites are met.

///

/// Any number of courses can be taken at the same time.

/// Return the minimum number of months needed to complete all the courses.

///

/// Note: The test cases are generated such that it is possible to complete

/// every course (i.e., the graph is a directed acyclic graph).

///

/// Example 1:

/// Input: n = 3, relations = [[1,3],[2,3]], time = [3,2,5]

/// Output: 8

/// Explanation: The figure above represents the given graph and the time

/// required to complete each course.

/// We start course 1 and course 2 simultaneously at month 0.

/// Course 1 takes 3 months and course 2 takes 2 months to complete

/// respectively.

/// Thus, the earliest time we can start course 3 is at month 3, and the

/// total time required is 3 + 5 = 8 months.

///

/// Example 2:

/// Input: n = 5, relations = [[1,5],[2,5],[3,5],[3,4],[4,5]],

/// time = [1,2,3,4,5]

/// Output: 12

/// Explanation: The figure above represents the given graph and the time

/// required to complete each course.

/// You can start courses 1, 2, and 3 at month 0.

/// You can complete them after 1, 2, and 3 months respectively.

/// Course 4 can be taken only after course 3 is completed, i.e., after 3

/// months. It is completed after 3 + 4 = 7 months.

/// Course 5 can be taken only after courses 1, 2, 3, and 4 have been

/// completed, i.e., after max(1,2,3,7) = 7 months.

/// Thus, the minimum time needed to complete all the courses

/// is 7 + 5 = 12 months.

/// Constraints:

/// 1. 1 <= n <= 5 \* 10^4

/// 2. 0 <= relations.length <= min(n \* (n - 1) / 2, 5 \* 10^4)

/// 3. relations[j].length == 2

/// 4. 1 <= prevCoursej, nextCoursej <= n

/// 5. prevCoursej != nextCoursej

/// 6. All the pairs [prevCoursej, nextCoursej] are unique.

/// 7. time.length == n

/// 8. 1 <= time[i] <= 10^4

/// 9. The given graph is a directed acyclic graph.

/// </summary>

int LeetCodeGraph::minimumTime(int n, vector<vector<int>>& relations, vector<int>& time)

{

vector<int> degree(n);

vector<int> start\_time(n);

vector<vector<int>> next\_courses(n);

for (size\_t i = 0; i < relations.size(); i++)

{

int prev = relations[i][0] - 1;

int next = relations[i][1] - 1;

next\_courses[prev].push\_back(next);

degree[next]++;

}

queue<int> waiting;

for (int i = 0; i < n; i++)

{

if (degree[i] == 0) waiting.push(i);

}

int result = 0;

while (!waiting.empty())

{

int course = waiting.front();

waiting.pop();

int finish\_time = start\_time[course] + time[course];

result = max(result, finish\_time);

for (size\_t j = 0; j < next\_courses[course].size(); j++)

{

int next\_course = next\_courses[course][j];

degree[next\_course]--;

start\_time[next\_course] = max(start\_time[next\_course], finish\_time);

if (degree[next\_course] == 0)

{

waiting.push(next\_course);

}

}

}

return result;

}

## 1857. Largest Color Value in a Directed Graph

Hard

There is a **directed graph** of n colored nodes and m edges. The nodes are numbered from 0 to n - 1.

You are given a string colors where colors[i] is a lowercase English letter representing the **color** of the ith node in this graph (**0-indexed**). You are also given a 2D array edges where edges[j] = [aj, bj] indicates that there is a **directed edge** from node aj to node bj.

A valid **path** in the graph is a sequence of nodes x1 -> x2 -> x3 -> ... -> xk such that there is a directed edge from xi to xi+1 for every 1 <= i < k. The **color value** of the path is the number of nodes that are colored the **most frequently** occurring color along that path.

Return *the****largest color value****of any valid path in the given graph, or*-1*if the graph contains a cycle*.

**Example 1:**

A picture containing text, clock

Description automatically generated

**Input:** colors = "abaca", edges = [[0,1],[0,2],[2,3],[3,4]]

**Output:** 3

**Explanation:** The path 0 -> 2 -> 3 -> 4 contains 3 nodes that are colored "a" (red in the above image).

**Example 2:**

Diagram, venn diagram

Description automatically generated

**Input:** colors = "a", edges = [[0,0]]

**Output:** -1

**Explanation:** There is a cycle from 0 to 0.

**Constraints:**

* n == colors.length
* m == edges.length
* 1 <= n <= 105
* 0 <= m <= 105
* colors consists of lowercase English letters.
* 0 <= aj, bj < n

### Analysis:

We use topology sort to process graph, the count the color in each node is the maximum from their predecessors.

/// <summary>

/// Leet code 1857. Largest Color Value in a Directed Graph

///

/// Hard

///

/// There is a directed graph of n colored nodes and m edges. The nodes

/// are numbered from 0 to n - 1.

///

/// You are given a string colors where colors[i] is a lowercase English

/// letter representing the color of the ith node in this graph

/// (0-indexed). You are also given a 2D array edges where edges[j] =

/// [aj, bj] indicates that there is a directed edge from node aj to node

/// bj.

///

/// A valid path in the graph is a sequence of nodes x1 -> x2 ->

/// x3 -> ... -> xk such that there is a directed edge from xi to xi+1

/// for every 1 <= i < k. The color value of the path is the number of

/// nodes that are colored the most frequently occurring color along that

/// path.

///

/// Return the largest color value of any valid path in the given graph,

/// or -1 if the graph contains a cycle.

///

/// Example 1:

/// Input: colors = "abaca", edges = [[0,1],[0,2],[2,3],[3,4]]

/// Output: 3

/// Explanation: The path 0 -> 2 -> 3 -> 4 contains 3 nodes that are

/// colored "a" (red in the above image).

///

/// Example 2:

/// Input: colors = "a", edges = [[0,0]]

/// Output: -1

/// Explanation: There is a cycle from 0 to 0.

///

/// Constraints:

/// 1. n == colors.length

/// 2. m == edges.length

/// 3. 1 <= n <= 10^5

/// 4. 0 <= m <= 10^5

/// 5. colors consists of lowercase English letters.

/// 6. 0 <= aj, bj < n

/// </summary>

int LeetCodeGraph::largestPathValue(string colors, vector<vector<int>>& edges)

{

int n = colors.size();

vector<vector<int>> neighbors(n);

vector<vector<int>> max\_colors(n, vector<int>(26));

vector<int> degree(n);

for (size\_t i = 0; i < edges.size(); i++)

{

neighbors[edges[i][0]].push\_back(edges[i][1]);

degree[edges[i][1]] ++;

}

queue<int> queue;

int count = 0;

for (int i = 0; i < n; i++)

{

if (degree[i] == 0) queue.push(i);

}

int result = 0;

while (!queue.empty())

{

int node = queue.front();

queue.pop();

count++;

max\_colors[node][colors[node] - 'a']++;

result = max(result, max\_colors[node][colors[node] - 'a']);

for (size\_t i = 0; i < neighbors[node].size(); i++)

{

int next = neighbors[node][i];

degree[next]--;

if (degree[next] == 0)

{

queue.push(next);

}

for (size\_t j = 0; j < 26; j++)

{

max\_colors[next][j] = max(max\_colors[next][j], max\_colors[node][j]);

}

}

}

if (count < n) return -1;

else return result;

}

## 1494. Parallel Courses II

Hard

You are given an integer n, which indicates that there are n courses labeled from 1 to n. You are also given an array relations where relations[i] = [prevCoursei, nextCoursei], representing a prerequisite relationship between course prevCoursei and course nextCoursei: course prevCoursei has to be taken before course nextCoursei. Also, you are given the integer k.

In one semester, you can take **at most** k courses as long as you have taken all the prerequisites in the **previous** semester for the courses you are taking.

Return *the****minimum****number of semesters needed to take all courses*. The testcases will be generated such that it is possible to take every course.

**Example 1:**

**A picture containing text, pool ball, sport, clipart

Description automatically generated**

**Input:** n = 4, dependencies = [[2,1],[3,1],[1,4]], k = 2

**Output:** 3

**Explanation:** The figure above represents the given graph.

In the first semester, you can take courses 2 and 3.

In the second semester, you can take course 1.

In the third semester, you can take course 4.

**Example 2:**

**A picture containing text, pool ball

Description automatically generated**

**Input:** n = 5, dependencies = [[2,1],[3,1],[4,1],[1,5]], k = 2

**Output:** 4

**Explanation:** The figure above represents the given graph.

In the first semester, you can take courses 2 and 3 only since you cannot take more than two per semester.

In the second semester, you can take course 4.

In the third semester, you can take course 1.

In the fourth semester, you can take course 5.

**Example 3:**

**Input:** n = 11, dependencies = [], k = 2

**Output:** 6

**Constraints:**

* 1 <= n <= 15
* 1 <= k <= n
* 0 <= relations.length <= n \* (n-1) / 2
* relations[i].length == 2
* 1 <= prevCoursei, nextCoursei <= n
* prevCoursei != nextCoursei
* All the pairs [prevCoursei, nextCoursei] are **unique**.
* The given graph is a directed acyclic graph.

### Analysis:

We use topology sort to scan the course by two passes, first we scan from back to front (starting from the course that no one depends on it), in this pass we can see each course’s depth, i.e. how many more dependency layers from this course. This means if we have multiple courses can be taken, we start the one with highest depth, so it has potential to free up more deeper courses.

The second pass is the simple toplogy with the help from priority queue with most depth first.

/// <summary>

/// Leet code #1494. Parallel Courses II

///

/// Hard

///

/// Given the integer n representing the number of courses at some

/// university labeled from 1 to n, and the array dependencies where

/// dependencies[i] = [xi, yi] represents a prerequisite relationship,

/// that is, the course xi must be taken before the course yi. Also,

/// you are given the integer k.

///

/// In one semester you can take at most k courses as long as you have

/// taken all the prerequisites for the courses you are taking.

///

/// Return the minimum number of semesters to take all courses. It is

/// guaranteed that you can take all courses in some way.

///

/// Example 1:

/// Input: n = 4, dependencies = [[2,1],[3,1],[1,4]], k = 2

/// Output: 3

/// Explanation: The figure above represents the given graph. In this

/// case we can take courses 2 and 3 in the first semester, then take

/// course 1 in the second semester and finally take course 4 in the

/// third semester.

///

/// Example 2:

/// Input: n = 5, dependencies = [[2,1],[3,1],[4,1],[1,5]], k = 2

/// Output: 4

/// Explanation: The figure above represents the given graph. In this

/// case one optimal way to take all courses is: take courses 2 and 3

/// in the first semester and take course 4 in the second semester, then

/// take course 1 in the third semester and finally take course 5 in the

/// fourth semester.

///

/// Example 3:

/// Input: n = 11, dependencies = [], k = 2

/// Output: 6

/// Constraints:

/// 1. 1 <= n <= 15

/// 2. 1 <= k <= n

/// 3. 0 <= dependencies.length <= n \* (n-1) / 2

/// 4. dependencies[i].length == 2

/// 5. 1 <= xi, yi <= n

/// 6. xi != yi

/// 7. All prerequisite relationships are distinct, that is,

/// dependencies[i] != dependencies[j].

/// 8. The given graph is a directed acyclic graph.

/// </summary>

int LeetCodeGraph::minNumberOfSemesters(int n, vector<vector<int>>& dependencies, int k)

{

vector<queue<int>> matrix1(n + 1), matrix2(n + 1);

vector<int> degree1(n + 1), degree2(n+1);

vector<int> depths(n + 1);

for (size\_t i = 0; i < dependencies.size(); i++)

{

degree1[dependencies[i][0]]++;

degree2[dependencies[i][1]]++;

matrix1[dependencies[i][1]].push(dependencies[i][0]);

matrix2[dependencies[i][0]].push(dependencies[i][1]);

}

queue<int> search;

for (int i = 1; i <= n; i++)

{

if (degree1[i] == 0) search.push(i);

}

int depth = 0;

while (!search.empty())

{

size\_t size = search.size();

for (size\_t i = 0; i < size; i++)

{

int course = search.front();

search.pop();

while (!matrix1[course].empty())

{

int next = matrix1[course].front();

matrix1[course].pop();

degree1[next]--;

if (degree1[next] == 0) search.push(next);

}

depths[course] = depth;

}

depth++;

}

int result = 0;

priority\_queue<pair<int, int>> pq;

for (int i = 1; i <= n; i++)

{

if (degree2[i] == 0) pq.push(make\_pair(depths[i], i));

}

while (!pq.empty())

{

result++;

queue<pair<int, int>> next\_queue;

for (int i = 0; i < k; i++)

{

if (pq.empty()) break;

pair<int, int> pos = pq.top();

pq.pop();

while (!matrix2[pos.second].empty())

{

int next = matrix2[pos.second].front();

matrix2[pos.second].pop();

degree2[next]--;

if (degree2[next] == 0) next\_queue.push(make\_pair(depths[next], next));

}

}

while (!next\_queue.empty())

{

pair<int, int> pos = next\_queue.front();

next\_queue.pop();

pq.push(pos);

}

}

return result;

}

## 269. Alien Dictionary

Hard

There is a new alien language that uses the English alphabet. However, the order among the letters is unknown to you.

You are given a list of strings words from the alien language's dictionary, where the strings in words are **sorted lexicographically** by the rules of this new language.

Return *a string of the unique letters in the new alien language sorted in****lexicographically increasing order****by the new language's rules. If there is no solution, return*""*. If there are multiple solutions, return****any of them***.

A string s is **lexicographically smaller** than a string t if at the first letter where they differ, the letter in s comes before the letter in t in the alien language. If the first min(s.length, t.length) letters are the same, then s is smaller if and only if s.length < t.length.

**Example 1:**

**Input:** words = ["wrt","wrf","er","ett","rftt"]

**Output:** "wertf"

**Example 2:**

**Input:** words = ["z","x"]

**Output:** "zx"

**Example 3:**

**Input:** words = ["z","x","z"]

**Output:** ""

**Explanation:** The order is invalid, so return "".

**Constraints:**

* 1 <= words.length <= 100
* 1 <= words[i].length <= 100
* words[i] consists of only lowercase English letters.

### Analysis:

We compare two words in dictionary, for the first characters different, we determine the sequence. Please remember we may have duplicate sequence, we should dedup use set.

We then use topology sort starting with no predecesors.

/// <summary>

/// Leet code #269. Alien Dictionary

/// There is a new alien language which uses the latin alphabet. However, the

/// order among letters are unknown to you. You receive a list of words from

/// the dictionary, where words are sorted lexicographically by the rules of

/// this new language. Derive the order of letters in this language.

///

/// For example,

/// Given the following words in dictionary,

/// [

/// "wrt",

/// "wrf",

/// "er",

/// "ett",

/// "rftt"

/// ]

/// The correct order is: "wertf".

/// Note:

/// 1.You may assume all letters are in lowercase.

/// 2.If the order is invalid, return an empty string.

/// 3.There may be multiple valid order of letters, return any one of them is

/// fine.

/// </summary>

string LeetCodeGraph::alienOrder(vector<string>& words)

{

string result;

vector<unordered\_set<int>> next\_set(26);

vector<int> degree(26, -1);

queue<int> search\_queue;

for (size\_t i = 0; i < words.size(); i++)

{

for (char ch : words[i])

{

if (degree[ch - 'a'] == -1) degree[ch - 'a'] = 0;

}

if (i > 0)

{

string prev\_word = words[i - 1];

string curr\_word = words[i];

for (size\_t i = 0; i < max(prev\_word.size(), curr\_word.size()); i++)

{

if (i == prev\_word.size()) break;

else if (i == curr\_word.size()) return "";

else

{

int prev = prev\_word[i] - 'a';

int curr = curr\_word[i] - 'a';

if (prev\_word[i] != curr\_word[i])

{

// this is required otherwise degree will be non-zero.

if (next\_set[prev].count(curr) == 0)

{

next\_set[prev].insert(curr);

degree[curr]++;

}

// when we hit one character difference, no point compare remaing.

break;

}

}

}

}

}

// remove all characters has predecessor

for (size\_t i = 0; i < degree.size(); i++)

{

if (degree[i] == 0)

{

search\_queue.push(i);

}

}

while (!search\_queue.empty())

{

int index = search\_queue.front();

search\_queue.pop();

for (char next : next\_set[index])

{

degree[next]--;

if (degree[next] == 0)

{

search\_queue.push(next);

}

}

result.push\_back(index + 'a');

}

for (size\_t i = 0; i < degree.size(); i++)

{

if (degree[i] > 0) return "";

}

return result;

}