# Topology Sort

The topology sort is based on BFS, a node may have some depdency, we start from the nodes with zero dependencies, and visit them first, and then we clear up the dependency on the those who depending on the visited nodes and select the next batch of nodes with zero dependency. When the search queue is empty if we still see the nodes not visited, this means we have circular dependency which can never be cleared up.

## 207. Course Schedule

Medium

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?

**Example 1:**

**Input:** 2, [[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:** 2, [[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.

**Note:**

1. The input prerequisites is a graph represented by **a list of edges**, not adjacency matrices. Read more about [how a graph is represented](https://www.khanacademy.org/computing/computer-science/algorithms/graph-representation/a/representing-graphs).
2. You may assume that there are no duplicate edges in the input prerequisites.

/// <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 LeetCode::canFinishCourse(int numCourses, vector<pair<int, 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 (pair<int, int> pair : prerequisites)

{

if (dependency[pair.second].count(pair.first) == 0)

{

degree[pair.first]++;

dependency[pair.second].insert(pair.first);

}

}

// 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);

}

// 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);

}

}

}

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

// all courses

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

{

if (degree[i] > 0) return false;

}

return true;

}

## 261. Graph Valid Tree

Medium

70622FavoriteShare

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.

**Example 1:**

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

**Output:** true

**Example 2:**

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

**Output:** false

**Note**: 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.

/// <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 LeetCode::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;

}

## 269. Alien Dictionary

Hard

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 **non-empty**words from the dictionary, where **words are sorted lexicographically by the rules of this new language**. Derive the order of letters in this language.

**Example 1:**

**Input:**

[

"wrt",

"wrf",

"er",

"ett",

"rftt"

]

**Output:** "wertf"

**Example 2:**

**Input:**

[

"z",

"x"

]

**Output:** "zx"

**Example 3:**

**Input:**

[

"z",

"x",

"z"

]

**Output:** ""

**Explanation:** The order is invalid, so return "".

**Note:**

1. You may assume all letters are in lowercase.
2. You may assume that if a is a prefix of b, then a must appear before b in the given dictionary.
3. If the order is invalid, return an empty string.
4. There may be multiple valid order of letters, return any one of them is fine.

/// <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 LeetCode::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;

}

## 582. Kill Process

Medium

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>

/// 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> LeetCode::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

41964FavoriteShare

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 Kso 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.

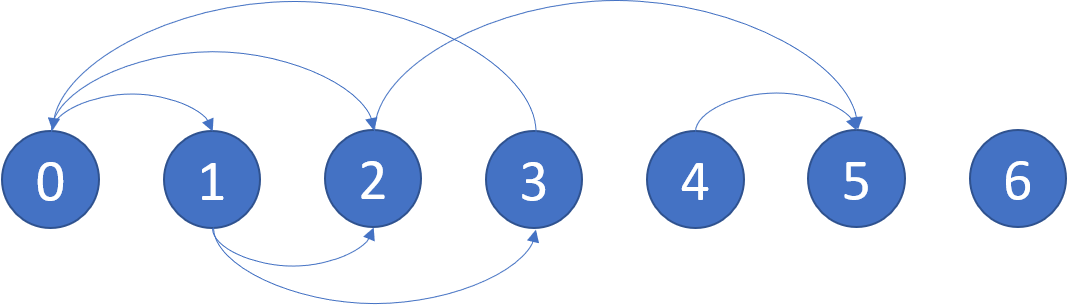
The directed graph has N nodes with labels 0, 1, ..., N-1, where N is the length of graph.  The graph is given in the following form: graph[i] is a list of labels j such that (i, j) is a directed edge of the graph.

**Example:**

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

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

Here is a diagram of the above graph.



**Note:**

* graph will have length at most 10000.
* The number of edges in the graph will not exceed 32000.
* Each graph[i] will be a sorted list of different integers, chosen within the range [0, graph.length - 1].

/// <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> LeetCode::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

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>

/// 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> LeetCode::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

Hard

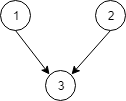
There are N courses, labelled from 1 to N.

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:**

****

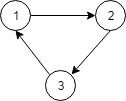
**Input:** N = 3, relations = [[1,3],[2,3]]

**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>

/// 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 LeetCode::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;

}