1136. Parallel Courses **Topological Sort** Medium Graph **Leetcode Link** 

### **Problem Description** The problem presents a scenario where you have n courses numbered from 1 to n. You are also given a list of pairs, relations,

representing the prerequisite relationships between the courses. Each pair [prevCourse, nextCourse] indicates that prevCourse must be completed before nextCourse can be taken.

The goal is to determine the minimum number of semesters required to complete all n courses, given that you can take any number of courses in a single semester, but you must have completed their prerequisites in the previous semester. If it's not possible to complete all courses due to a circular dependency or any other reason, the result should be -1.

To solve this problem efficiently, we use a graph-based approach known as Topological Sorting, which is usually applied to

### scheduling tasks or finding a valid order in which to perform tasks with dependencies. Here's a step-by-step breakdown of the intuition behind the solution:

Intuition

1. We consider each course as a node in a directed graph where an edge from node prevCourse to node nextCourse indicates that prevCourse is a prerequisite for nextCourse.

2. We construct an in-degree list to track the number of prerequisite courses for each course. A course with an in-degree of 0

be available for the next semester.

the number of prerequisites for each course.

that can be taken in the first semester as they have no prerequisites.

A defaultdict for constructing the adjacency list graph representation.

courses are only considered available once all their prerequisites have been fulfilled.

before taking course 2, course 2 before course 3, and course 3 before course 4.

The algorithms and patterns used are part of graph theory and include:

- means it has no prerequisite and can be taken immediately. 3. We then use a queue to keep track of all courses that have no prerequisites (in-degree of 0). This set represents the courses we
- can take in the current semester. 4. For each semester, we dequeue all available courses (i.e., those with no prerequisites remaining) and decrease the in-degree of
- their corresponding next courses by 1, representing that we have taken one of their prerequisites. 5. If the in-degree of these next courses drops to 0, it means they have no other prerequisites left, so we add them to the queue to
- 6. We continue this process, semester by semester, keeping a counter to track the number of semesters necessary until there are no courses left to take.
- dependency, and we return -1. Otherwise, we return the counter, which now holds the minimum number of semesters needed to take all courses.

7. If at the end of this process, there are still courses left (i.e., n is not zero), it implies that there is a cycle or unsatisfiable

**Solution Approach** 

The implementation follows the topological sorting approach using the Kahn's algorithm pattern. Here's how the algorithm unfolds:

1. Graph Representation: The graph is represented using an adjacency list, g, which maps each course to a list of courses that

depend on it (i.e., the courses that have it as a prerequisite). Additionally, an in-degree list, indeg, is maintained to keep track of

# 2. Initialization: An initial scan of the relations array is performed to populate the adjacency list and in-degree list. Each relation

[prev, next] increments the in-degree of the next course to indicate an additional prerequisite. 3. Queue of Available Courses: A queue, q, is initialized with all the courses that have an in-degree of 0. These are the courses

- 4. Iterating Over Semesters: We use a while loop to simulate passing semesters. In each iteration, we process all courses in the queue: Increment the ans variable, which represents the number of semesters.
- Update the in-degree of all "child" courses (i.e., courses that the current course is a prerequisite for) by decrementing their respective in-degree count.

5. Detecting Cycles or Infeasible Conditions: After processing all possible courses, if the n counter is not zero, it suggests that not

all courses can be taken due to cycles (circular prerequisites) or other unsolvable dependencies. In such a case, we return -1.

- 6. Returning the Result: If the loop completes successfully and n is zero, the ans variable contains the minimum number of semesters needed to take all courses, which is then returned.
- A list to maintain in-degree counts. A deque to efficiently add and remove available courses as we iterate over semesters.

∘ For each course taken (dequeued), decrement n to reflect that one course has been completed.

∘ If any "child" course's in-degree becomes ∅, enqueue it, meaning it can be taken in the next semester.

 Topological sorting to determine a valid order to take the courses. Kahn's algorithm for detecting cycles and performing the sort iteratively by removing nodes with zero in-degree.

Throughout the process, we maintain a balance between the courses taken and the updates to the graph's state, ensuring that

Suppose we have n = 4 courses and the relations list as [[1,2], [2,3], [3,4]]. According to this list, you must complete course 1

Let's walk through a small example to illustrate the solution approach.

enqueue it for the next semester.

1 from collections import defaultdict, deque

prereq -= 1

course -= 1

semesters += 1

semesters = 0

incoming\_degree = [0] \* total\_courses

for prereq, course in prerequisites:

graph[prereq].append(course)

incoming\_degree[course] += 1

for \_ in range(len(queue)):

course = queue.popleft()

# Populate the graph and update the incoming degree counts

# Adjust indices to be 0-based instead of 1-based

# Initialize a variable to count the number of semesters

# Process all nodes in the current level

for next\_course in graph[course]:

return semesters if total\_courses == 0 else -1

public int minimumSemesters(int n, int[][] relations) {

Arrays.setAll(graph, k -> new ArrayList<>());

int minimumSemesters(int N, vector<vector<int>>& relations) {

// Build the graph and fill in the in-degree array

// Add edge from prevCourse to nextCourse

// Increment the in-degree of the nextCourse

graph[prevCourse].push\_back(nextCourse);

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

for (auto& relation : relations) {

++inDegree[nextCourse];

for (int i = 0; i < N; ++i) {

if (inDegree[i] == 0) {

queue.push(i);

while (!queue.empty()) {

queue.pop();

++semesters;

queue<int> queue;

// Queue to perform topological sorting

// Perform the topological sort using BFS

int course = queue.front();

--N; // One less course to consider

for (int nextCourse : graph[course]) {

queue.push(nextCourse);

if (--inDegree[nextCourse] == 0) {

vector<int> inDegree(N);

// Graph representation where each node points to its successors

int prevCourse = relation[0] - 1; // Convert to 0-indexed

int nextCourse = relation[1] - 1; // Convert to 0-indexed

// Enqueue all the courses that have no prerequisites (in-degree of 0)

// Process all nodes in the current level of the queue

int semesters = 0; // Number of semesters needed to complete all the courses

for (int levelSize = queue.size(); levelSize > 0; --levelSize) {

// Check all the next courses that current course points to

function minimumSemesters(totalCourses: number, prerequisitePairs: number[][]): number {

// Initialize an array to keep track of how many prerequisites each course has

const courseGraph = Array.from({ length: totalCourses }, () => []);

// Queue for courses that have no prerequisites, hence can be taken

// Find the initial courses with no prerequisites and add them to the queue

// Process all courses that are available to be taken in this semester

const currentCourse = queue.shift(); // Take the course (remove from queue)

queue.push(nextCourse); // If no more prerequisites, add to the queue

// If all courses have been taken, return the total number of semesters needed, otherwise return -1

--totalCourses; // Decrement the count of courses remaining to be taken

const inDegree = new Array(totalCourses).fill(0);

inDegree[course - 1]++;

if (inDegree[index] === 0) {

semesters++; // Start a new semester

return totalCourses === 0 ? semesters : -1;

queue.push(index);

const queue: number[] = [];

while (queue.length > 0) {

Time and Space Complexity

for (const [prerequisite, course] of prerequisitePairs) {

courseGraph[prerequisite - 1].push(course - 1);

for (let index = 0; index < totalCourses; ++index) {</pre>

// Create an adjacency list to represent the course graph, initialized with empty arrays

// Build the graph and fill in the inDegree array by iterating over the prerequisitePairs

// Decrement the in-degree and if it drops to 0, add it to the queue

// Increment the semester count as all the nodes in the queue can be taken in the current semester

// In-degree array to keep track of the number of prerequisites for each course

List<Integer>[] graph = new List[n];

int[] indegree = new int[n];

// Initialize each list within 'graph'.

incoming\_degree[next\_course] -= 1

queue.append(next\_course)

if incoming\_degree[next\_course] == 0:

// Create an adjacency list to represent the graph structure.

// Array to keep track of in-degrees (number of prerequisites).

# Initialize a queue with all courses having 0 incoming degree

**Example Walkthrough** 

immediately.

Python Solution

9

10

11

12

13

14

15

16

17

18

19

20

21

22

27

28

29

30

31

32

33

34

35

41

42

43

6

8

9

from typing import List

The data structures employed in this solution are:

• Graph Representation: We construct a graph such that g = {1: [2], 2: [3], 3: [4]}, and the in-degree list indeg = [0, 1, 1, 1] (since the numbering starts at 1, you might need to adjust the indices accordingly).

Using the approach described above, let's proceed step by step:

Iterating Over Semesters:

Second Semester: We dequeue course 2 from q and decrement n to 2. We update the in-degree of course 3 to 0 and

Detecting Cycles or Infeasible Conditions: Through our iterations, we didn't encounter any cycles, and all indegree counts

• First Semester: We dequeue course 1 and decrement n to 3 (as we have one less course to complete). We then decrease the

• Third Semester: Course 3 is dequeued, n is decremented to 1, the in-degree of course 4 is now 0, and it gets enqueued for

• Initialization: We perform an initial scan and find that course 1 has no prerequisites (in-degree = 0), so it can be taken

• Queue of Available Courses: We initialize a queue q with course 1 in it, ready to be taken in the first semester.

the next semester. ∘ Fourth Semester: Finally, we dequeue course 4, and n is now 0, which means all courses have been accounted for.

could be reduced to 0 properly, indicating that we can plan courses without conflict.

in-degree of course 2 to 0 and enqueue it for the next semester.

- Returning the Result: We've successfully queued and completed all courses, one per semester. Therefore, the minimum number of semesters required is 4, which is the value of our counter after the last course is taken. The result is 4 semesters to complete the given courses following the pre-defined prerequisites-order.
  - class Solution: def minimumSemesters(self, total\_courses: int, prerequisites: List[List[int]]) -> int: # Create a default dictionary to hold the adjacency list representation of the graph 6 graph = defaultdict(list) # Initialize a list for incoming degree count of each node 8
- 23 24 # Process nodes level by level (BFS) 25 while queue: 26 # Increment the semester count

total\_courses -= 1 # Decrement the number of courses remaining

# If adjacent course has no more prerequisites, add it to the queue

# it means there's a cycle, and we return -1 to indicate it's not possible to finish

# Decrease the incoming degree of all adjacent courses

queue = deque([i for i, degree in enumerate(incoming\_degree) if degree == 0])

```
36
37
38
           # If all courses are covered, return the number of semesters
39
           # If there are still courses remaining with nonzero incoming degree,
40
```

Java Solution

C++ Solution

2 public:

6

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

58 };

59

6

8

9

10

11

12

13

14

15

16

17

18

19

20

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

46

45 }

1 class Solution {

class Solution {

```
// Build the graph and count the in-degrees for each course.
10
11
            for (int[] relation : relations) {
12
                int prevCourse = relation[0] - 1; // Convert to 0-based index.
13
                int nextCourse = relation[1] - 1; // Convert to 0-based index.
14
                graph[prevCourse].add(nextCourse);
15
                indegree[nextCourse]++;
16
17
18
            // Queue to store courses with no prerequisites (in-degree of 0).
            Deque<Integer> queue = new ArrayDeque<>();
19
20
            // Enqueue all courses with no prerequisites.
21
            for (int i = 0; i < n; ++i) {
22
                if (indegree[i] == 0) {
23
                    queue.offer(i);
24
25
26
27
            // Variable to keep count of the minimum number of semesters.
28
            int semesters = 0;
29
            // Perform BFS to process courses.
30
            while (!queue.isEmpty()) {
31
                semesters++; // Increase semester count.
                // Process courses in the current semester.
33
                for (int k = queue.size(); k > 0; --k) {
                    // Dequeue the front course.
34
35
                    int course = queue.poll();
36
                    n--; // Decrease the count of the remaining courses.
                    // Decrease in-degree for the subsequent courses and check if they become ready to take.
37
                    for (int nextCourse : graph[course]) {
                        if (--indegree[nextCourse] == 0) {
39
40
                            queue.offer(nextCourse);
41
42
43
44
45
46
           // If there are no more courses to take, return semesters, otherwise return -1 to indicate the graph is not acyclic.
47
            return n == 0 ? semesters : -1;
48
49
50
```

#### 52 53 54 55 // If all courses are processed, we return the number of semesters, otherwise it's impossible (-1) return N == 0 ? semesters : -1; 57

Typescript Solution

#### 21 22 23 24 // "ans" will keep track of the total number of semesters required 25 let semesters = 0; 26 27 // Perform a breadth-first search (BFS) on the course graph

for (let count = queue.length; count > 0; --count) {

if (--inDegree[nextCourse] === 0) {

for (const nextCourse of courseGraph[currentCourse]) {

// Decrement the in-degree of the adjacent courses

- gives a complexity of O(n). 3. The while loop to reduce the indegrees and find the order of courses: Within the loop, each vertex is processed exactly once, as it gets removed from the queue, and each edge is considered exactly once, as it gets decremented when the previous vertex is
- processed. So, the complexity related to processing all vertices and edges is 0(V + E), where V is the number of vertices and E is the number of edges.
- The space complexity of the code:
- 3. The queue q could potentially store all n vertices in the worst case, which gives a space complexity of O(n).

E could be as large as n \* (n-1) in a dense graph, we do not multiply by n in the space complexity as we consider distinct elements

1. The graph g has a space complexity of O(E), as it stores all the edges. 2. The indegree array indeg has a space complexity of O(n) as it stores an entry for each vertex.

As such, the overall space complexity is the sum of the complexities of g, indeg, and q, which gives O(E + n). Note that even though

Thus, the overall time complexity is O(E) + O(n) + O(V + E). Since V = n, we can simplify this to O(n + E).

## The time complexity of the provided code can be analyzed step by step: 1. Creating the graph g and the indegree array indeg: This involves iterating over each relation once. If there are E relations, this operation has a complexity of O(E). 2. Building the initial queue q with vertices of indegree 0: This involves a single scan through the indeg array of n elements, which

Time Complexity

within the data structure.

**Space Complexity**