In this problem, you are tasked with finding the minimum number of months needed to complete a set of courses with certain

prerequisites. There are n courses labeled from 1 to n. A 2D integer array relations is given, where each pair [prevCourse,

Problem Description

nextCourse] signifies that the prevCourse must be completed before the nextCourse can be taken. You also have an array time representing the number of months required to complete each course, indexed at 0. There are two key rules you must follow: you can start any course at any time as long as its prerequisites are satisfied, and you can take as many courses concurrently as you desire. The task is to calculate the minimum total time needed to complete all courses, under the constraint that a direct acyclic graph (DAG) represents the course structure. Intuition

approach, where we: 1. Determine the prerequisites (in-degree) for each course. 2. Create a graph representing the course dependencies. 3. Use a queue to process courses that have no prerequisites.

To solve this problem, we need to process courses in a way that respects prerequisite relations and keeps track of the total time

needed to complete each course individually, as well as the overall time to finish all courses. This lends itself to a topological sort

- 5. Use a variable (ans) to maintain the maximum finishing time across all courses.
- The provided solution iterates through the courses, initializing a queue with those that have no prerequisites and setting their finishing time in the array f using the time array. We then go through the courses in the queue one by one, updating the finishing

A graph g is represented as a dictionary of lists, mapping each course to its list of next courses.

4. Keep an array (f) to store the finishing time for each course, based on its prerequisites' finishing times.

time for their dependent courses and ensuring we take the maximum time between the time already recorded for that course and the finishing time of the current course plus the time required for the dependent course. We decrement the in-degree of each

dependent course and add it to the queue if its in-degree drops to zero.

Using these steps, we guarantee that we process the courses such that prerequisites are always completed before dependent courses and that concurrency is maximized, as there are no restrictions on the number of courses that can be taken simultaneously. The final answer (ans) represents the minimum number of months needed to complete all courses, as it accounts for the longest path through the graph, which effectively is the critical path that determines the overall time to completion. Solution Approach

The solution to this problem efficiently makes use of topological sorting, a common algorithm for processing Directed Acyclic Graphs (DAGs), alongside dynamic programming to calculate minimum times.

• An in-degree array indeg, which keeps track of how many prerequisites each course has. A queue q, implemented as a deque, used for the topological sort.

Algorithm Steps:

Data Structures:

An array f, used to store the finishing time for each course, or the minimum months needed to reach that course.

1. Graph and In-Degree Initialization: For each relation in relations, the graph g and the in-degree array indeg are populated.

2. Identifying Start Courses: We iterate over all courses. Those without prerequisites (indeg[i] == 0) are added to the queue q,

Every relation [a, b] indicates b is dependent on a, so a is added to the list of b's prerequisites in the graph, and b's in-degree is

and their finishing time f[i] is set to their respective time from the time array.

incremented since it has one more course that precedes it.

added to the queue for processing.

Let's illustrate the solution approach using a small example:

Initialize the graph g and in-degree array indeg.

= [0, 0, 2, 1] will have the count of prerequisites for each course.

its prerequisites.

3. Topological Sort and Finishing Time Calculation:

While the queue is not empty:

Dequeue an element i. For each course j that i is a prerequisite for: Update the finishing time for course j (f[j]) to be the maximum of the current stored time and the sum of f[i] and the time to complete course j. This ensures f[j] reflects the total time taken to reach course j after completing all

■ Decrement the in-degree of course j. If j's in-degree becomes 0, it means all its prerequisites are met, and it's

 Update the overall answer ans to be the maximum of its current value and f[j]. 4. Result: After processing all courses, ans holds the minimum number of months required to complete all courses. This is because ans tracks the longest path (in months) through the graph, which corresponds to the time it would take to finish the courses if they were taken sequentially following their dependencies.

The algorithm successfully computes the minimum time to finish all courses by leveraging topological sort to respect course

prerequisites and dynamic programming to calculate the minimum finish times, taking advantage of the ability to take multiple

concurrent nature of course enrollment. **Example Walkthrough**

Imagine a scenario with 4 courses, where the time array is given as time = [1, 2, 3, 1], which indicates that course 1 takes 1

month to complete, course 2 takes 2 months, and so on. The relations array is given as relations = [[1, 3], [2, 3], [3, 4]],

courses concurrently. This solution ensures that all courses are processed in the correct order and that the time tracking reflects the

meaning course 1 and 2 are prerequisites for course 3, and course 3 is a prerequisite for course 4.

Conclusion:

2. Identifying Start Courses: • We see that course 1 and course 2 have zero prerequisites (indeg[0] == 0 and indeg[1] == 0). Initialize the queue q with these courses and set f = [1, 2, 0, 0], where f[0] = 1 and f[1] = 2. 3. Topological Sort and Finishing Time Calculation:

For relations = [[1, 3], [2, 3], [3, 4]], we build the graph g = {1: [3], 2: [3], 3: [4]}. The in-degree array indeg

Decrease in-degree of course 3 (indeg[2] becomes 1). Since in-degree of course 3 is not 0, it remains in the graph.

Process the courses in the queue:

time[2]) = $\max(0, 1 + 3) = 4$.

time[3]) = $\max(0, 5 + 1) = 6$.

1. Graph and In-Degree Initialization:

f[1] + time[2]) = max(4, 2 + 3) = 5.■ Decrease in-degree of course 3 (indeg[2] becomes 0), and add course 3 to the queue. ■ Dequeue course 3, with finishing time of 5 months. Course 4 is a dependent, so update f[3] to max(f[3], f[2] +

Next, dequeue course 2, with a finishing time of 2 months. Again, course 3 is a dependent, so update f[2] to max(f[2],

Dequeue course 1, with finishing time of 1 month. Course 3 is a dependent, so update f[2] to max(f[2], f[0] +

concurrently, and respecting the prerequisite requirements.

the proposed solution to compute the minimum total time required to complete all the courses.

Create a graph using a dictionary and initialize in-degrees for all nodes.

graph[start - 1].append(end - 1) # Adjusting indices to be 0-based.

finish_times[i] = duration # The finish time is the task's own duration.

max_time = max(max_time, duration) # Check if this is the new max time.

If the dependent task has no more prerequisites, add it to the queue.

Return the global maximum time, which is the minimum time to complete all tasks.

Build the graph and update in-degrees based on prerequisites.

Initialize a queue for processing tasks with no prerequisites.

max_time = max(max_time, finish_times[dependent])

Decrease the in-degree of the dependent task.

This will keep track of the global maximum time.

Initialize an array to keep track of the finish times of each task.

■ Decrease in-degree of course 4 (indeg[3] becomes 0), and add course 4 to the queue.

• The queue is now empty, and we can calculate the maximum time from the array f, which is ans = 6 months.

4. Result: We conclude that it will take a minimum of 6 months to complete all courses, given the ability to take multiple courses

This walk-through illustrates the process of topological sorting, dynamic programming, and concurrent course enrollment utilized by

Python Solution 1 from collections import defaultdict, deque from typing import List

def minimumTime(self, total_tasks: int, prerequisites: List[List[int]], task_durations: List[int]) -> int:

20 max_time = 0 21 22 # Enqueue tasks with no prerequisites and set their finish times. for i, (in_deg, duration) in enumerate(zip(in_degree, task_durations)): 23 if in_deg == 0: 24

```
32
               # Iterate through the current task's dependent tasks.
33
                for dependent in graph[current_task]:
34
                    # Calculate the finish time of the dependent task.
35
                    finish\_times[dependent] = max(finish\_times[dependent], finish\_times[current\_task] + task\_durations[dependent])
36
                    # Update the global maximum time.
```

class Solution:

8

9

10

11

12

13

14

15

16

17

18

19

25

26

27

28

29

30

31

37

38

39

40

41

42

43

44

49

50

51

52

53

55

54 }

C++ Solution

2 public:

3

8

9

10

11

12

13

14

15

16

17

1 class Solution {

graph = defaultdict(list)

queue = deque()

while queue:

in_degree = [0] * total_tasks

for start, end in prerequisites:

in degree [end - 1] += 1

finish_times = [0] * total_tasks

queue.append(i)

Process tasks in the queue.

current_task = queue.popleft()

in_degree[dependent] -= 1

if in_degree[dependent] == 0:

queue.append(dependent)

// Return the maximum time to finish all courses

// Construct the graph and fill in-degree data

graph[prevCourse].push_back(nextCourse);

// Increment in-degree for the 'nextCourse'

// Create an edge from 'prevCourse' to 'nextCourse'

for (const auto& relation : relations) {

int prevCourse = relation[0] - 1;

int nextCourse = relation[1] - 1;

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

++inDegree[nextCourse];

vector<int> inDegree(n, 0);

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

// Create a graph data structure to represent courses and their prerequisites

// A vector to keep track of the number of prerequisites (in-degree) for each course

// Courses are numbered starting 1, so subtract 1 to get a 0-based index

return maxTime;

```
45
            return max_time
46
Java Solution
   class Solution {
       public int minimumTime(int totalCourses, int[][] prerequisites, int[] timeToComplete) {
           // Create a graph represented as an adjacency list
           List<Integer>[] graph = new List[totalCourses];
           Arrays.setAll(graph, k -> new ArrayList<>());
           // Create an array to track the number of prerequisites for each course
8
           int[] incomingEdges = new int[totalCourses];
9
10
           // Build the graph and count incoming edges for each node (course)
11
12
           for (int[] relation : prerequisites) {
               // Courses are 1-indexed, but our array/graph is 0-indexed
13
               int prerequisite = relation[0] - 1;
14
                int course = relation[1] - 1;
15
16
               graph[prerequisite].add(course);
               ++incomingEdges[course];
17
18
19
20
           // Initialize a queue for courses that have no prerequisites
21
           Deque<Integer> queue = new ArrayDeque<>();
22
           // Array to store the time to finish each course including prerequisites
23
           int[] finishTime = new int[totalCourses];
24
           // Variable to store the maximum time to finish all courses
25
           int maxTime = 0;
26
27
           // Enqueue courses without prerequisites and set their finish times
28
           for (int i = 0; i < totalCourses; ++i) {</pre>
                if (incomingEdges[i] == 0) {
29
                   queue.offer(i);
30
                    finishTime[i] = timeToComplete[i];
31
32
                   maxTime = Math.max(maxTime, finishTime[i]);
33
34
35
36
           // Process the courses in topological order
37
           while (!queue.isEmpty()) {
               int currentCourse = queue.pollFirst();
38
               // For each course that follows the current one
39
                for (int nextCourse : graph[currentCourse]) {
40
                   // Update the finish time to the maximum of the current known time and the time required after finishing the current
41
                    finishTime[nextCourse] = Math.max(finishTime[nextCourse], finishTime[currentCourse] + timeToComplete[nextCourse]);
42
                   maxTime = Math.max(maxTime, finishTime[nextCourse]);
43
                   // If the course has no remaining prerequisites, add it to the queue
44
                    if (--incomingEdges[nextCourse] == 0) {
45
                        queue.offer(nextCourse);
46
47
48
```

31 32 33 34

```
18
 19
 20
             // Queue for BFS traversal
 21
             queue<int> courseQueue;
 22
             // Vector to keep track of the total time to complete each course
 23
             vector<int> finishTime(n, 0);
 24
             // Initialize the final answer to time completion
 25
             int totalTime = 0;
 26
 27
             // Enqueue courses with no prerequisites
 28
             for (int i = 0; i < n; ++i) {
 29
                 if (inDegree[i] == 0) {
 30
                     courseQueue.push(i);
                     finishTime[i] = time[i];
                     // Update total time with the time taken to complete this course
                     totalTime = max(totalTime, time[i]);
 35
 36
 37
             // Process courses in the order they can be taken
 38
             while (!courseQueue.empty()) {
 39
                 int currentCourse = courseQueue.front();
 40
                 courseQueue.pop();
 41
 42
                 // Check all the next courses that can be taken after the current course
 43
                 for (int nextCourse : graph[currentCourse]) {
 44
                     // Decrease the in-degree of the next course
 45
                     --inDegree[nextCourse];
 46
                     // If all prerequisites of the next course have been taken, add to queue
 47
                     if (inDegree[nextCourse] == 0) {
 48
                         courseQueue.push(nextCourse);
 49
 50
 51
 52
                     // Update the finish time of the next course
 53
                     // It will be the maximum of its current finish time and
                     // the finish time of its prerequisite (current course) plus its own duration
 54
 55
                     finishTime[nextCourse] = max(finishTime[nextCourse], finishTime[currentCourse] + time[nextCourse]);
 56
                     // Update the total time completion with the updated finish time of the next course
                     totalTime = max(totalTime, finishTime[nextCourse]);
 57
 58
 59
 60
             // Return the maximum time required to finish all courses
 61
 62
             return totalTime;
 63
 64 };
 65
Typescript Solution
     function minimumTime(numCourses: number, prerequisites: number[][], courseDurations: number[]): number {
         // Graph represented as an adjacency list
         const graph: number[][] = Array(numCourses)
             .fill(0)
             .map(() => []);
  6
        // Array to keep track of incoming edges (in-degree) for each node
         const inDegrees: number[] = Array(numCourses).fill(0);
  9
         // Build graph and populate inDegrees array
 10
 11
         for(const [source, target] of prerequisites) {
             graph[source - 1].push(target - 1);
 12
 13
             ++inDegrees[target - 1];
 14
 15
```

35 36 37 38

return maxTime;

Time and Space Complexity

const queue: number[] = [];

let maxTime: number = 0;

for(let i = 0; i < numCourses; ++i) {</pre>

// Array to keep the completion time for each course

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

// Return the maximum time to complete all courses

queue.push(nextCourse);

const finishTimes: number[] = Array(numCourses).fill(0);

// Variable to keep track of the maximum time required to finish all courses

maxTime = Math.max(maxTime, finishTimes[nextCourse]);

16

17

18

19

20

21

22

23

24

25

26

39

40

41

42

43

44

45

46

47

48

50

49 }

27 if(inDegrees[i] === 0) { 28 queue.push(i); 29 finishTimes[i] = courseDurations[i]; maxTime = Math.max(maxTime, finishTimes[i]); 30 31 32 33 // Process courses in topological order 34 while(queue.length > 0) { const currentCourse = queue.shift()!; for(const nextCourse of graph[currentCourse]) { finishTimes[nextCourse] = Math.max(finishTimes[nextCourse], finishTimes[currentCourse] + courseDurations[nextCourse]);

// Queue for processing courses that have no prerequisites or whose prerequisites have been completed

// Initialize the queue with courses that have no prerequisites and calculate their finish times

// When a prerequisite course is completed, decrease in-degree of the next course

The time complexity of the given code is O(V + E), where V is the number of courses (nodes) and E is the number of dependencies (edges). This complexity arises because each course is enqueued and dequeued exactly once, which is O(V). Every dependency is looked at exactly once when iterating through the adjacency list, which gives O(E). Therefore, the total is O(V + E). The space complexity is also 0(V + E) because we need to store the graph g, which contains E edges, the indeg array of size V, and

the f array of size V. Considering the queue's space, in the worst case, it could hold all vertices, which adds up to 0(V). However, since this does not change the overall space complexity, it remains O(V + E).