Solution: Course Schedule

Let's solve the Course Schedule problem using the Topological Sort pattern.

We'll cover the following

Statement

There are a total of `numCourses` courses you have to take. The courses are labeled from `0` to `numCourses - 1`. You are also given a `prerequisites` array, where `prerequisites[i] = [a[i], b[i]]` indicates that you must take course `b[i]` first if you want to take the course `a[i]`. For example, the pair $[1, 0]$ indicates that to take course 1, you have to first take course 0.

Return TRUE if all of the courses can be finished. Otherwise, return FALSE.

Constraints:

- $1 \leq$ `numCourses` ≤ 2000
- $0 \leq$ `prerequisites.length` ≤ 5000
- `prerequisites[i].length` $= 2$
- $0 \leq$ `a[i]`, `b[i]` $<$ `numCourses`
- All the pairs `prerequisites[i]` are unique.

Solution

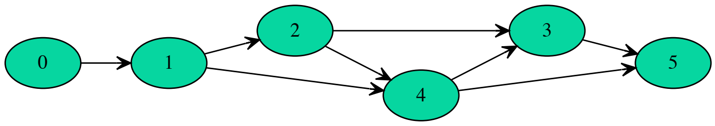
Initialize the hash map with the vertices and their children. We'll use another hash map to keep track of the number of in-degrees of each vertex. Then we'll find the source vertex (with 0 in-degree) and increment the `counter`. Retrieve the source node's children and add them to the queue. Decrement the in-degrees of the retrieved children. We'll check whether the in-degree of the child vertex becomes equal to zero, and we'll increment the `counter`. Repeat the process until the queue is empty.

> **Note:** The in-degree is the number of edges coming into a vertex in a directed graph.

The primary purpose of finding a vertex with 0 in-degree is to find a course with a pre-requisite count of 0. When we take a course, say a (that is the pre-requisite of another course, say b), we'll decrement the in-degree of b by 1, and if the in-degree count becomes 0, we can say that the b's pre-requisites have been completed.

The slide deck below illustrates the algorithm above, where `numCourses = 6`:

Find out in-degree of each vertex and remove vertex containing
0 in-degree one by one.



We can see the code of this solution below:

Java

```java
class CourseSchedule {
    public static boolean canFinish(int numCourses, int[][] prerequisites) {
        int counter = 0;
        if (numCourses <= 0)
          return false;

        // Initialize the graph
        // count of incoming prerequisites
        HashMap<Integer, Integer> inDegree = new HashMap<>();
        HashMap<Integer, List<Integer>> graph = new HashMap<>();
        for (int i = 0; i < numCourses; i++) {
            inDegree.put(i, 0);
            graph.put(i, new ArrayList<Integer>());
        }

        // b. Build the graph
        for (int i = 0; i < prerequisites.length; i++) {
            int parent = prerequisites[i][0], child = prerequisites[i][1];
            graph.get(parent).add(child); // put the child into it's parent's list
            inDegree.put(child, inDegree.get(child) + 1); // increment child's inDegree
```

```java
        Queue<Integer> sources = new LinkedList<>();
        for (Map.Entry<Integer, Integer> entry : inDegree.entrySet()) {
            if (entry.getValue() == 0)
                sources.add(entry.getKey());
```

Course Schedule

Time complexity

In the algorithm above, each course will become a source only once, and each edge will be accessed and removed once. Therefore, the above algorithm's time complexity will be $O(V + E)$, where V is the total number of vertices and E is the total number of edges in the graph.

Space complexity

The space complexity will be $O(V + E)$ because we're storing all of the edges for each vertex in an adjacency list.