<https://leetcode.com/problems/course-schedule/>

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 <= 2000
* 0 <= prerequisites.length <= 5000
* prerequisites[i].length == 2
* 0 <= ai, bi < numCourses
* All the pairs prerequisites[i] are **unique**.

**Attempt 1: 2022-11-20**

**Solution 1:  Detect Cycle in a Directed Graph using BFS [Topological Sort ] (10min)**

**Style 1 (Classic: Build graph + Get indegree based on graph + Topological Sort with BFS)**

class Solution {

public boolean canFinish(int numCourses, int[][] prerequisites) {

// Build graph

Map<Integer, List<Integer>> graph = new HashMap<Integer, List<Integer>>();

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

graph.put(i, new ArrayList<Integer>());

}

for(int[] pre : prerequisites) {

graph.get(pre[1]).add(pre[0]);

}

// Get indegree based on graph

Map<Integer, Integer> indegree = new HashMap<Integer, Integer>();

for(int node : graph.keySet()) {

indegree.put(node, 0);

}

for(int node : graph.keySet()) {

for(int neighbour : graph.get(node)) {

indegree.put(neighbour, indegree.get(neighbour) + 1);

}

}

// Topological Sort

Queue<Integer> q = new LinkedList<Integer>();

for(int node : indegree.keySet()) {

if(indegree.get(node) == 0) {

q.offer(node);

}

}

// 'count' for recording number of courses can take

int count = 0;

while(!q.isEmpty()) {

int node = q.poll();

count++;

for(int neighbour : graph.get(node)) {

indegree.put(neighbour, indegree.get(neighbour) - 1);

if(indegree.get(neighbour) == 0) {

q.offer(neighbour);

}

}

}

return count == numCourses;

}

}

Time Complexity: O(n+m)

The overall time complexity of this basic algorithm is O(n+m). The O(n) comes from the number of times that the while loop (and initial for loop) is executed, and the O(m) from the nested for loop.

Although there is no way to calculate how many times the inner loop will be executed on any one iteration of the outer loop, it will only be executed once for each successor of each member, which means that the total number of times that it will be executed is the total number of successors of all the members -- or the total number of relations.

Space Complexity: O(n+m)

Space complexity is also O(n+m). The O(n) component comes from the predecessor count information stored for each member, and the maximum length of the auxiliary queue. The O(m) comes from storing the successors for each member; once again, the total number of successors is the number of relations, so O(m).

**Style 2 (More elegant)**

class Solution {

public boolean canFinish(int numCourses, int[][] prerequisites) {

// Get indegree

int[] indegree = new int[numCourses];

for(int[] pre : prerequisites) {

indegree[pre[0]]++;

}

// Topological Sort

Queue<Integer> q = new LinkedList<Integer>();

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

if(indegree[i] == 0) {

q.offer(i);

}

}

while(!q.isEmpty()) {

int node = q.poll();

numCourses--;

for(int[] pre : prerequisites) {

if(pre[1] == node) {

indegree[pre[0]]--;

if(indegree[pre[0]] == 0) {

q.offer(pre[0]);

}

}

}

}

return numCourses == 0;

}

}

Time Complexity: O(n+m)

The overall time complexity of this basic algorithm is O(n+m). The O(n) comes from the number of times that the while loop (and initial for loop) is executed, and the O(m) from the nested for loop.

Although there is no way to calculate how many times the inner loop will be executed on any one iteration of the outer loop, it will only be executed once for each successor of each member, which means that the total number of times that it will be executed is the total number of successors of all the members -- or the total number of relations.

Space Complexity: O(n+m)

Space complexity is also O(n+m). The O(n) component comes from the predecessor count information stored for each member, and the maximum length of the auxiliary queue. The O(m) comes from storing the successors for each member; once again, the total number of successors is the number of relations, so O(m).

**Solution 2:  Detect Cycle in a Directed Graph using DFS [Backtracking] (10min)**

**Style 1: DFS helper method name as 'hasCycle', when cycle detected return true**

class Solution {

public boolean canFinish(int numCourses, int[][] prerequisites) {

boolean[] visited = new boolean[numCourses];

boolean[] recursionStack = new boolean[numCourses];

// Build graph

Map<Integer, List<Integer>> graph = new HashMap<Integer, List<Integer>>();

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

graph.put(i, new ArrayList<Integer>());

}

for(int[] pre : prerequisites) {

graph.get(pre[1]).add(pre[0]);

}

// DFS to find cycle based on each node

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

// Additional check 'visited' or not to promote

// efficiency on the fly since 'visited' will be

// update during DFS

if(!visited[i]) {

// If we have cycle then not able to finish

if(hasCycle(i, graph, visited, recursionStack)) {

return false;

}

}

}

return true;

}

private boolean hasCycle(int course, Map<Integer, List<Integer>> graph, boolean[] visited, boolean[] recursionStack) {

// If course detect on same path again then find cycle

if(recursionStack[course]) {

return true;

}

if(visited[course]) {

return false;

}

visited[course] = true;

// Backtrack the current path since switch to other path

recursionStack[course] = true;

for(int neighbour : graph.get(course)) {

if(hasCycle(neighbour, graph, visited, recursionStack)) {

return true;

}

}

recursionStack[course] = false;

return false;

}

}

Time Complexity: O(n+m)

Space Complexity: O(n+m)

**Style 2: DFS helper method name as 'noCycle', when cycle detected return false, exactly reverse logic than Style 1**

class Solution {

public boolean canFinish(int numCourses, int[][] prerequisites) {

boolean[] visited = new boolean[numCourses];

boolean[] recursionStack = new boolean[numCourses];

// Build graph

Map<Integer, List<Integer>> graph = new HashMap<Integer, List<Integer>>();

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

graph.put(i, new ArrayList<Integer>());

}

for(int[] pre : prerequisites) {

graph.get(pre[1]).add(pre[0]);

}

// DFS to find cycle based on each node

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

// Additional check 'visited' or not to promote

// efficiency on the fly since 'visited' will be

// update during DFS

if(!visited[i]) {

// If we have cycle then not able to finish

if(!noCycle(i, graph, visited, recursionStack)) {

return false;

}

}

}

return true;

}

private boolean noCycle(int course, Map<Integer, List<Integer>> graph, boolean[] visited, boolean[] recursionStack) {

// If course detect on same path again then find cycle

if(recursionStack[course]) {

return false;

}

if(visited[course]) {

return true;

}

visited[course] = true;

// Backtrack the current path since switch to other path

recursionStack[course] = true;

for(int neighbour : graph.get(course)) {

if(!noCycle(neighbour, graph, visited, recursionStack)) {

return false;

}

}

recursionStack[course] = false;

return true;

}

}

Time Complexity: O(n+m)

Space Complexity: O(n+m)

**Graph Algorithm - Cycle Detection in Directed Graph using DFS**

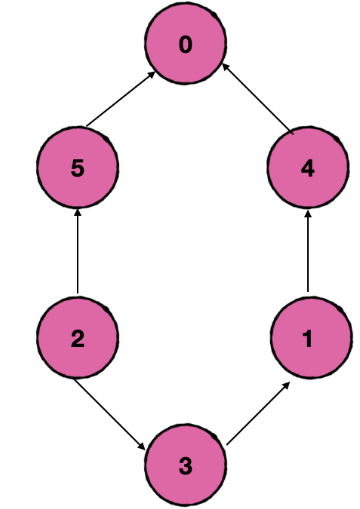
**Refer to**

<https://dev.to/rohithv07/graph-algorithm-cycle-detection-in-directed-graph-using-dfs-4bl5>

**What is a Cycle**

In graph theory, a path that starts from a given node and ends on the same node is a cycle.

**Cycle Detection in an Directed Graph**



A directed graph is a set of objects, otherwise called vertices or nodes, connected together and all the edges are directed from one vertex to another. A directed graph is an ordered pair G = (V, E) where, V is a set of elements known as vertices or nodes. E is a set of ordered pair of vertices called as edges or directed edges.

Cycle in a directed graph can be detected with the help of Depth-First Search algorithm.

DFS Algorithm for Cycle Detection in an Directed Graph

The dfs algorithm for cycle detection in undirected graph will not work here because we cannot say that directed graph is having a cycle, if we get to a node which is already marked as visited and previous node is different.

📌 Initialize a visited boolean array with all nodes unvisited, a boolean **recursion stack** with all nodes set to false.

A **recursion stack** is to track the nodes that are currently in recursion. We mark node as true if the node has further recursion calls and change it to false for no recursion calls.

📌 Run a loop from 0 to n - 1 as the graph may have different components.

📌 If the current node is not visited, call the dfs recursive function passing the current node, visited array, **recursion stack** array.

dfs(graph, node, visited, recursionStack)

📌 Inside the dfs function, check if the node is already in the **recursion stack**.

If it is already in the **recursion stack**, this means we are going to repeat the recursion call which results in a cycle. So we detect cycle in graph and return true.

📌 Check if the node is already visited. If yes, return false.

📌 Mark the node as visited and mark the node in **recursion stack**.

📌 Traverse through the children of the current node.

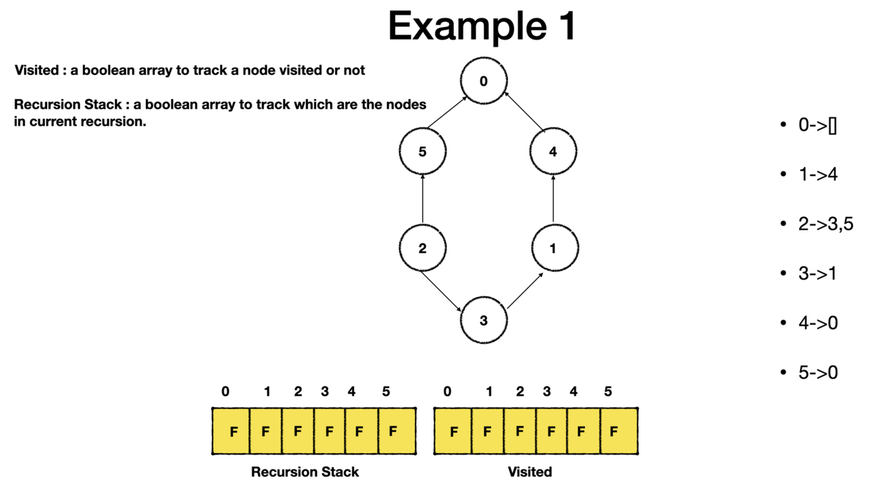
📌 Continue doing the recursion for all the children.

📌 If the recursion calls for the current node is over, reset the value to false in the **recursion stack** array.

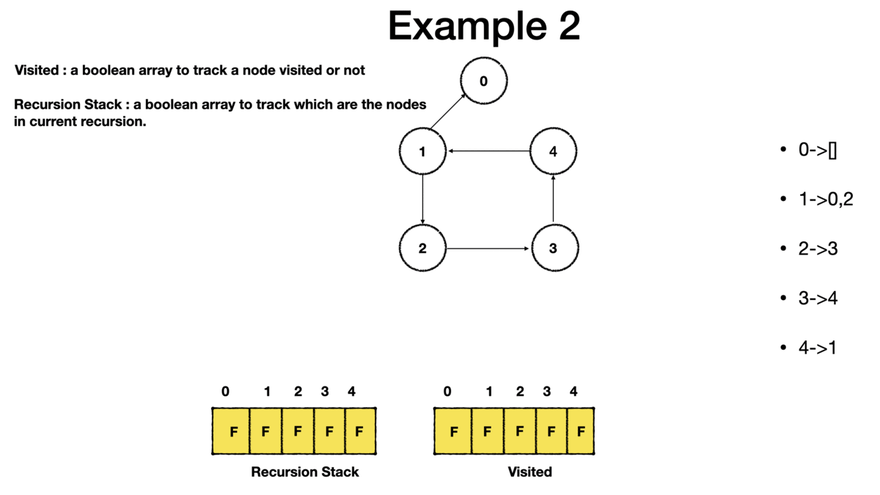
📌 If we get out of the initial for loop and all the nodes are now visited, this means we have no cycle.

**Example**

1. **A Directed Graph with No Cycle.**



1. **A Directed Graph with Cycle.**



**Time and Space Complexity**

* We are traversing through all the nodes and edges. So time complexity will be O(V + E) where V = vertices or node, E = edges.
* We use a visited array, **recursion stack** array and an adjacency list for the graph. So the space complexity will be O(V) + O(V) + O(V + E) + extra space for the recursion calls.



**Another different Coloring Algorithm for DFS**

**Refer to**

<https://leetcode.com/problems/course-schedule/discuss/58564/detailed-comment-on-dfs-solution-for-course-schedule-i-ii>

For the DFS part, I can understand now. Thanks for sharing! For more details, please refer to classic CLRS.

The 0, 1, 2 in visited array corresponds to White, Gray, Black in CLRS. As CLRS says, **The key idea is that when we first explore an edge, the color of vertex tells us something about the edge:**

1. WHITE (vis[nb] == 0) indicates a tree edge,
2. GRAY (vis[nb] == 1) indicates a back edge, and
3. BLACK (vis[nb] == 2) indicates a forward or cross edge.

public boolean canFinish(int n, int[][] prereq) {

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

for (int i = 0; i < n; i++) adj[i] = new ArrayList<>();

for (int[] e : prereq) adj[e[1]].add(e[0]);

int[] vis = new int[n]; // reuse so that each node visited once

for (int i = 0; i < n; i++) // must check every node. eg.[1,0],[0,1]

if (dfs(adj, i, vis)) return false;

return true;

}

// Check if back edge (directed cycle) exists. If not => DAG => able to topo sort

private boolean dfs(List<Integer>[] adj, int v, int[] vis) {

vis[v] = 1;

for (int nb : adj[v]) {

if (vis[nb] == 1) return true; // visited and nb is v's ancestor => back edge

if (vis[nb] == 0 && dfs(adj, nb, vis)) return true; // nb is not visited => tree edge

// else vis[nb]==2, nb is visited but not ancestor => forward or cross edge

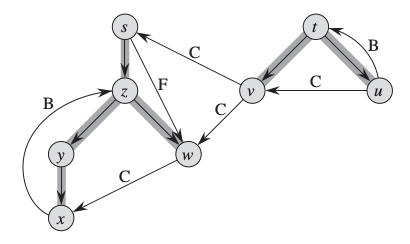
}

vis[v] = 2;

return false;

}

The illustration below is an example. F for forward edge, C for cross edge, B for back edge.



Using same idea, we can solve Course Schedule II in DFS manner as well. The key note is **Topological Sort order is the reverse order of their finish time**.

public int[] findOrder(int n, int[][] prereq) {

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

for (int i = 0; i < n; i++) adj[i] = new ArrayList<>();

for (int[] e : prereq) adj[e[1]].add(e[0]);

LinkedList<Integer> ret = new LinkedList<>();

int[] vis = new int[n];

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

if (vis[i] == 0 && dfs(ret, adj, i, vis)) return new int[0];

return ret.stream().mapToInt(i -> i).toArray();

}

private boolean dfs(LinkedList<Integer> ret, List<Integer>[] adj, int v, int[] vis) {

vis[v] = 1;

for (int nb : adj[v]) {

if (vis[nb] == 1) return true;

if (vis[nb] == 0 && dfs(ret, adj, nb, vis)) return true;

}

vis[v] = 2;

ret.addFirst(v); // Topo-sort order is the reverse order of their finish time

return false;

}

**Another explain refer to**

<https://leetcode.com/problems/course-schedule/discuss/658275/C%2B%2B-DFS-Easiest-Solution-With-Explanation-(My-1st-approach)>

We just have to find if our graph contains cycle or not because if graph contains cycle then all t nodes in cycle are interdependent and 1 course cannot be completed because its prerequisite is dependent on other course and it goes on .We used coloring algorithm to find if there is cycle in graph or not.

**Coloring Algorithm**

vis[id]=0 is used for node which is not yet visited

vis[id]=1 is used for the node which is visited and currently its child nodes are being visited

vis[id]=2 done when all the child nodes of a node ("id") are visited and the function returns to parent node of node ("id")

So at that time it is marked as 2 because this node does not require any further traversing.

bool iscycle(vector<int> adj[],vector<int> &vis,int id){

if(vis[id]==1)

return true;

if(vis[id]==0){

vis[id]=1;

for(auto edge : adj[id]){

if(iscycle(adj,vis,edge))

return true;

}

}

vis[id] = 2;

return false;

}

bool canFinish(int n, vector<vector<int>>& pre) {

vector<int> adj[n];

for(auto edge : pre)

adj[edge[1]].push\_back(edge[0]);

vector<int> vis(n,0);

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

if(iscycle(adj,vis,i))

return false;

}

return true;

}