LeetCode 207 — Course Schedule

🔗 https://leetcode.com/problems/course-schedule/

1. Problem Title & Link

Title: LeetCode 207: Course Schedule
Link: https://leetcode.com/problems/course-schedule/

2. Problem Statement (Short Summary)

Given numCourses and a list prerequisites where each pair [a, b] means *to take course a you must first take course b*, determine **if it is possible to finish all courses**.
Equivalently: does the directed graph of prerequisites contain a **cycle**? If yes → impossible; if no → possible.
Return true if you can finish all courses, otherwise false.

3. Examples (Input → Output)

Example 1

Input: numCourses = 2, prerequisites = [[1,0]]
Output: true
Explanation: Take course 0 first then 1.

Example 2

Input: numCourses = 2, prerequisites = [[1,0],[0,1]]
Output: false
Explanation: Cycle 0 → 1 → 0, impossible.

Example 3

Input: numCourses = 4, prerequisites = [[1,0],[2,0],[3,1],[3,2]]
Output: true
Explanation: Possible order: 0 → {1,2} → 3

4. Constraints

- Graph problem with V = numCourses, E = len(prerequisites).
- Aim for **O(V + E)** time and **O(V + E)** space.
- Typical platform limits vary (small to ~10^5), so prefer linear algorithms (DFS cycle detect or Kahn's BFS/ topological sort).

5. Core Concept (Pattern / Topic)

⭐ **Graph (Directed) — Cycle Detection / Topological Sort**

Subtopics: DFS with recursion-state, Kahn's algorithm (BFS using indegrees)

6. Thought Process (Step-by-Step Explanation)

Intuition

If the directed graph of prerequisites has a cycle, there's no way to order courses — return false. Otherwise, there exists a topological ordering — return true.

Two standard approaches

Approach A — DFS (Detect cycle via recursion-stack)

- For each node do DFS.
- Track states: 0 = unvisited, 1 = visiting (in recursion stack), 2 = visited.
- If during DFS you see a neighbor in visiting state → found a cycle → return false.
- If DFS finishes with no back-edge → true.

Approach B — Kahn's Algorithm (BFS / indegree)

- Compute indegree for all nodes.
- Initialize queue with nodes of indegree 0 (no prerequisites).
- Pop node, decrement indegree of neighbors; if neighbor becomes 0 push it.
- Count popped nodes; if count == numCourses → possible; else cycle exists → impossible.

Both are O(V+E). Kahn also directly produces an ordering if needed (Course Schedule II).

7. Visual / Intuition Diagram (ASCII)

Graph for [[1,0],[2,0],[3,1],[3,2]]:

0 → 1 → 3

0 → 2 → 3

Kahn:

indegree: 0:0, 1:1, 2:1, 3:2

queue: [0] -> pop 0 -> reduce 1,2 -> queue [1,2] -> pop 1 -> reduce 3 -> queue [2] -> pop 2 -> reduce 3 -> queue [3] -> pop 3 -> count=4 -> success

DFS states:

start 0: visiting -> neighbors processed -> visited

no back-edge to visiting nodes -> no cycle

8. Pseudocode (Language Independent)

DFS cycle-detect

```
state = [0]*n

function dfs(node):
    if state[node] == 1: return false    # found cycle
    if state[node] == 2: return true     # already safe
    state[node] = 1
    for nei in adj[node]:
```

```
        if not dfs(nei): return false
    state[node] = 2
    return true

for v in 0..n-1:
    if state[v] == 0:
        if not dfs(v): return false
return true
```

Kahn (BFS)

```
build adj list and indegree array
q = queue of nodes with indegree 0
count = 0
while q not empty:
    node = q.pop()
    count += 1
    for nei in adj[node]:
        indegree[nei] -= 1
        if indegree[nei] == 0: q.push(nei)
return count == numCourses
```

9. Code Implementation

✅ **Python (DFS cycle detection)**

```
from collections import defaultdict
class Solution:
    def canFinish(self, numCourses: int, prerequisites: List[List[int]]) -> bool:
        adj = defaultdict(list)
        for course, pre in prerequisites:
            adj[pre].append(course)

        state = [0] * numCourses  # 0=unvisited,1=visiting,2=visited

        def dfs(node):
            if state[node] == 1:
                return False
            if state[node] == 2:
                return True
            state[node] = 1
            for nei in adj[node]:
                if not dfs(nei):
                    return False
            state[node] = 2
```

```
            return True

        for v in range(numCourses):
            if state[v] == 0:
                if not dfs(v):
                    return False
        return True
```

✅ **Python (Kahn's BFS / indegree)**

```
from collections import defaultdict, deque
class Solution:
    def canFinish(self, numCourses: int, prerequisites: List[List[int]]) -> bool:
        adj = defaultdict(list)
        indeg = [0] * numCourses

        for course, pre in prerequisites:
            adj[pre].append(course)
            indeg[course] += 1

        q = deque([i for i in range(numCourses) if indeg[i] == 0])
        count = 0
        while q:
            node = q.popleft()
            count += 1
            for nei in adj[node]:
                indeg[nei] -= 1
                if indeg[nei] == 0:
                    q.append(nei)

        return count == numCourses
```

✅ **Java (DFS)**

```
class Solution {
    private int[] state; // 0=unvisited,1=visiting,2=visited
    private List<Integer>[] adj;

    public boolean canFinish(int numCourses, int[][] prerequisites) {
        adj = new ArrayList[numCourses];
        for (int i = 0; i < numCourses; i++) adj[i] = new ArrayList<>();
        for (int[] p : prerequisites) {
```

```
            adj[p[1]].add(p[0]);
        }
        state = new int[numCourses];
        for (int i = 0; i < numCourses; i++) {
            if (state[i] == 0 && !dfs(i)) return false;
        }
        return true;
    }

    private boolean dfs(int node) {
        if (state[node] == 1) return false;
        if (state[node] == 2) return true;
        state[node] = 1;
        for (int nei : adj[node]) {
            if (!dfs(nei)) return false;
        }
        state[node] = 2;
        return true;
    }
}
```

✅ **Java (Kahn's BFS)**

```
class Solution {
    public boolean canFinish(int numCourses, int[][] prerequisites) {
        List<Integer>[] adj = new ArrayList[numCourses];
        for (int i = 0; i < numCourses; i++) adj[i] = new ArrayList<>();
        int[] indeg = new int[numCourses];

        for (int[] p : prerequisites) {
            adj[p[1]].add(p[0]);
            indeg[p[0]]++;
        }

        Queue<Integer> q = new LinkedList<>();
        for (int i = 0; i < numCourses; i++)
            if (indeg[i] == 0) q.add(i);

        int count = 0;
        while (!q.isEmpty()) {
            int node = q.poll();
            count++;
            for (int nei : adj[node]) {
```

```
                if (--indeg[nei] == 0) q.add(nei);
            }
        }
        return count == numCourses;
    }
}
```

10. Time & Space Complexity

- **Time:** O(V + E) where V = numCourses, E = len(prerequisites) — both DFS and Kahn visit each node and edge once.
- **Space:** O(V + E) for adjacency list + recursion stack (DFS) or queue/indegree (BFS).

11. Common Mistakes / Edge Cases

- ❌ Using node values as indices incorrectly when courses not 0..n-1 (LeetCode uses 0..n-1)
- ❌ Forgetting that prerequisites are directed (order matters)
- ❌ Not handling self-loop [i,i] → cycle
- ❌ Counting duplicate prerequisite edges multiple times in indegree (usually fine but beware of multi-edges if input allows)
- Edge cases:
  - numCourses = 0 (trivial true)
  - No prerequisites → true
  - Self-loop or cycle → false
  - Disconnected graph components → handle via iterating all nodes

12. Detailed Dry Run (Step-by-Step Table)

Input: numCourses=4, prereq=[[1,0],[2,0],[3,1],[3,2]] (Kahn run)

Step	indegree array	queue	popped node	count
init	[0,1,1,2]	[0]	—	0
pop 0	[0,1,1,2]	[]	0	1
reduce 1,2	[0,0,0,2]	[1,2]	—	1
pop 1	[0,0,0,2]	[2]	1	2
reduce 3	[0,0,0,1]	[2]	—	2
pop 2	[0,0,0,1]	[3]	2	3
reduce 3	[0,0,0,0]	[3]	—	3
pop 3	[0,0,0,0]	[]	3	4

end	count == numCourses → success			TRUE

13. Common Use Cases (Real-Life / Interview)

- Scheduling tasks with dependencies (build systems, job scheduling)
- Package installation order resolution
- Topological ordering problems (course planning, pipeline execution)
- Detecting circular dependencies in systems

14. Common Traps (Important!)

- Confusing direction of edge (prerequisite -> course vs course -> prerequisite). For this problem pair [a,b] is edge b -> a.
- Using recursion without tracking visiting state leads to false negatives/positives.
- Not iterating over all nodes (miss disconnected components).

15. Builds To (Related LeetCode Problems)

- **LC 210** — Course Schedule II (return topological order)
- **LC 444 / 1462** — Graph scheduling variants
- **Topological sort problems** — build systems, task scheduling, build-order problems

16. Alternate Approaches + Comparison

Approach	Time	Space	Pros
DFS (recursion-state)	O(V+E)	O(V+E)	Simple, direct cycle detection
Kahn's BFS (indegree)	O(V+E)	O(V+E)	Also produces ordering if needed; iterative (no recursion)
Union-Find	Not suitable directly for directed cycle detection	—	Union-Find handles undirected cycles only — avoid here

17. Why This Solution Works (Short Intuition)

Finishing all courses is possible **iff** the prerequisite graph has no directed cycle. DFS detects a back-edge to the recursion stack (cycle). Kahn's algorithm peels off nodes with no prerequisites; if at the end some nodes remain, they're part of a cycle.

18. Variations / Follow-Up Questions

- Return a valid order (LC 210) — use Kahn and record popped sequence.
- Allow repeated prerequisites or invalid indices — sanitize input.
- Weighted tasks / time to complete courses with parallel execution (scheduling with durations).
- Detect and return the actual cycle nodes for diagnostics.