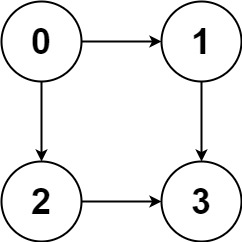
Given a directed acyclic graph (**DAG**) of n nodes labeled from 0 to n - 1, find all possible paths from node 0 to node n - 1 and return them in **any order**.

The graph is given as follows: graph[i] is a list of all nodes you can visit from node i (i.e., there is a directed edge from node i to node graph[i][j]).

**Example 1:**

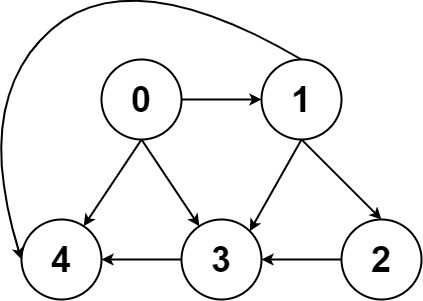


**Input:** graph = [[1,2],[3],[3],[]]

**Output:** [[0,1,3],[0,2,3]]

**Explanation:** There are two paths: 0 -> 1 -> 3 and 0 -> 2 -> 3.

**Example 2:**



**Input:** graph = [[4,3,1],[3,2,4],[3],[4],[]]

**Output:** [[0,4],[0,3,4],[0,1,3,4],[0,1,2,3,4],[0,1,4]]

**Example 3:**

**Input:** graph = [[1],[]]

**Output:** [[0,1]]

**Example 4:**

**Input:** graph = [[1,2,3],[2],[3],[]]

**Output:** [[0,1,2,3],[0,2,3],[0,3]]

**Example 5:**

**Input:** graph = [[1,3],[2],[3],[]]

**Output:** [[0,1,2,3],[0,3]]

Solution:

class Solution {

List<List<Integer>> result = new ArrayList<>();

public List<List<Integer>> allPathsSourceTarget(int[][] graph) {

List<Integer> path = new ArrayList<>();

dfs(graph, path, 0);

return result;

}

private void dfs(int[][] graph, List<Integer> path, int u) {

path.add(u);

if(u == graph.length - 1)

result.add(new ArrayList(path));

else{

for(int v : graph[u]){

dfs(graph, path, v);

path.remove(path.size() - 1);

}

}

}

}