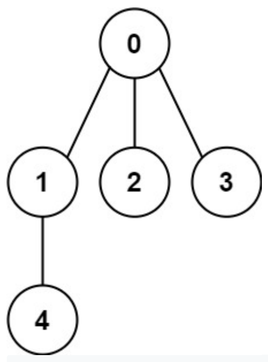
261. Graph Valid Tree

Medium 1923 52 Add to List Share

You have a graph of `n` nodes labeled from `0` to `n - 1`. You are given an integer n and a list of `edges` where `edges[i] = [a_i, b_i]` indicates that there is an undirected edge between nodes `a_i` and `b_i` in the graph.

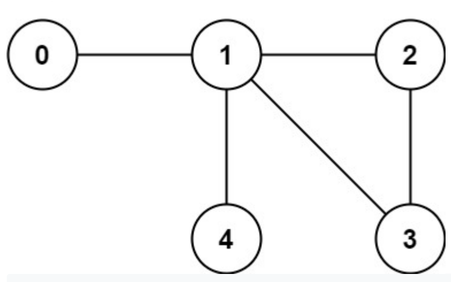
Return `true` *if the edges of the given graph make up a valid tree, and* `false` *otherwise.*

Example 1:




```
Input: n = 5, edges = [[0,1],[0,2],[0,3],[1,4]]
Output: true
```

Example 2:




```
Input: n = 5, edges = [[0,1],[1,2],[2,3],[1,3],[1,4]]
Output: false
```

Constraints:

- `1 <= n <= 2000`
- `0 <= edges.length <= 5000`
- `edges[i].length == 2`
- `0 <= a_i, b_i < n`
- `a_i != b_i`
- There are no self-loops or repeated edges.

Accepted 208,461 | Submissions 465,314

Seen this question in a real interview before? Yes No

Companies

0 ~ 6 months 6 months ~ 1 year 1 year ~ 2 years

LinkedIn | 6 Google | 5 Coupang | 3 Microsoft | 2 Bloomberg | 2

Related Topics

Depth-First Search Breadth-First Search Union Find Graph

Similar Questions

Course Schedule	Medium
Number of Connected Components in an Undirected Graph	Medium
Keys and Rooms	Medium

Hide Hint 1

Given `n = 5` and `edges = [[0, 1], [1, 2], [3, 4]]`, what should your return? Is this case a valid tree?

Hide Hint 2

According to the definition of tree on Wikipedia: “a tree is an undirected graph in which any two vertices are connected by *exactly* one path. In other words, any connected graph without simple cycles is a tree.”

Java Autocomplete

```java
class Solution {

    private List<List<Integer>> adjacencyList = new ArrayList<>();
    private Set<Integer> seen = new HashSet<>();


    public boolean validTree(int n, int[][] edges) {

        if (edges.length != n - 1) return false;

        // Make the adjacency list.
        for (int i = 0; i < n; i++) {
            adjacencyList.add(new ArrayList<>());
        }
        for (int[] edge : edges) {
            adjacencyList.get(edge[0]).add(edge[1]);
            adjacencyList.get(edge[1]).add(edge[0]);
        }

        // Carry out depth first search.
        dfs(0);
        // Inspect result and return the verdict.
        return seen.size() == n;
    }

    public void dfs(int node) {
        if (seen.contains(node)) return;
        seen.add(node);
        for (int neighbour : adjacencyList.get(node)) {
            dfs(neighbour);
        }
    }
}
```