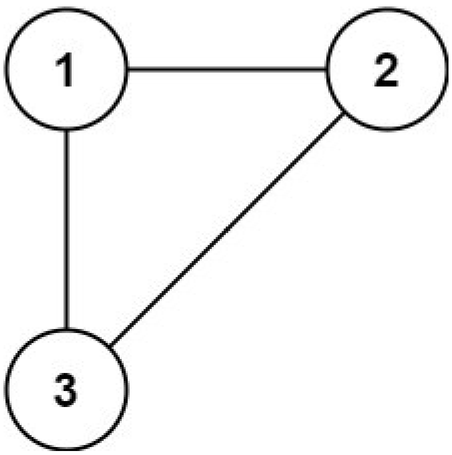
684. Redundant Connection

n this problem, a tree is an **undirected graph** that is connected and has no cycles.

You are given a graph that started as a tree with `n` nodes labeled from `1` to `n`, with one additional edge added. The added edge has two **different** vertices chosen from `1` to `n`, and was not an edge that already existed. The graph is represented as an array `edges` of length `n` where `edges[i] = [ai, bi]` indicates that there is an edge between nodes `ai` and `bi` in the graph.

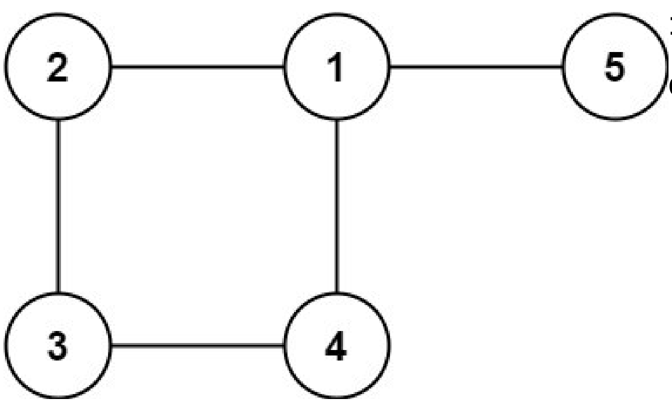
Return *an edge that can be removed so that the resulting graph is a tree of* `n` *nodes*. If there are multiple answers, return the answer that occurs last in the input.

Example 1:




```
Input: edges = [[1,2],[1,3],[2,3]]
Output: [2,3]
```

Example 2:




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

Constraints:

- `n == edges.length`
- `3 <= n <= 1000`
- `edges[i].length == 2`
- `1 <= ai < bi <= edges.length`
- `ai != bi`
- There are no repeated edges.
- The given graph is connected.

Overview

We are given a graph consisting of N nodes and $N-1$ edges, which means the graph initially forms a tree. A tree is a special type of graph that is connected (there is a path between any two nodes) and acyclic (it does not contain any cycles). However, a new edge is added to the tree, connecting two nodes that are already part of the graph. This new edge creates a cycle because there are now two distinct paths between some pairs of nodes. As a result, the graph is no longer a tree but a single-cycle graph.

Our goal is to identify the edge that, if removed, will restore the graph to its original state as a tree. Since the tree must be connected and acyclic, removing any edge from the cycle will break the cycle and turn the graph into a tree. However, if there are multiple edges that can be removed to achieve this, we are required to return the edge that appears last in the given list of edges.

684. Redundant Connection

```
/*
   Union Find: Kruskal's MST algorithm
   Time complexity: O(nα(n))
   Space complexity: O(3n)
   n: #node = #edges
*/
typedef std::vector<int> vi;
typedef std::vector<vi> vvi;
```

```
class DSU{
   public:
      vi parent;
      vi group;

      DSU(int _N){
         parent.resize(_N);
         group.resize(_N,1);
         for(int i=0;i<_N;++i){
            parent[i]=i;
         }
      }

      int find(int p){
         if(p==parent[p]) return p;
         return parent[p]=find(parent[p]);
      }

      bool unify(int p,int q){
         int parent_p=find(p);
         int parent_q=find(q);
         if(parent_p==parent_q) return false;

         if(group[parent_p]<group[parent_q]){
            group[parent_q]+=group[parent_p];
            group[parent_p]=1;
            parent[parent_p]=parent_q;

         }
         else{
            group[parent_p]+=group[parent_q];
            group[parent_q]=1;
            parent[parent_q]=parent_p;
         }

         return true;
      }
};
```

```
class Solution {
   public:
      vi findRedundantConnection(vvi& edges) {
         int n=edges.size();

         DSU dsu=DSU(n);

         for(auto& edge: edges){
            int u=--edge[0];
            int v=--edge[1];
            if(!dsu.unify(u,v)) return {++u,++v};
         }

         return {};
      }
};
```