685. Redundant Connection II

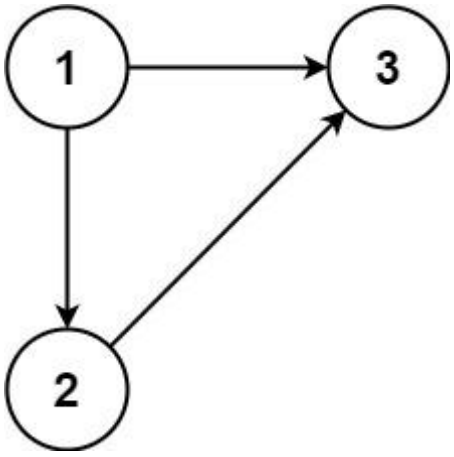
In this problem, a rooted tree is a directed graph such that, there is exactly one node (the root) for which all other nodes are descendants of this node, plus every node has exactly one parent, except for the root node which has no parents.

The given input is a directed graph that started as a rooted tree with `n` nodes (with distinct values from `1` to `n`), with one additional directed edge added. The added edge has two different vertices chosen from `1` to `n`, and was not an edge that already existed.

The resulting graph is given as a 2D-array of `edges`. Each element of `edges` is a pair `[ui, vi]` that represents a directed edge connecting nodes `ui` and `vi`, where `ui` is a parent of child `vi`.

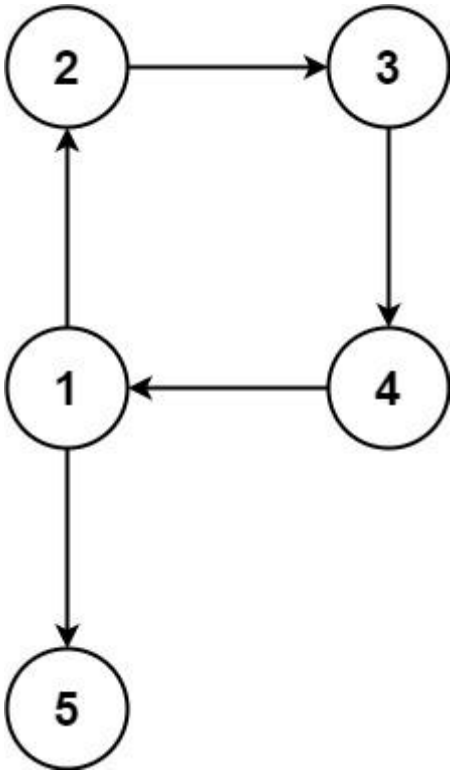
Return *an edge that can be removed so that the resulting graph is a rooted tree of* `n` *nodes*. If there are multiple answers, return the answer that occurs last in the given 2D-array.

Example 1:




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

Example 2:




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

Constraints:

- n == edges.length
- 3 <= n <= 1000
- edges[i].length == 2
- 1 <= u_i, v_i <= n
- u_i != v_i

Code:

```
class Solution:
    def findRedundantDirectedConnection(self, edges: List[List[int]]) -> List[int]:
```

```
rank = [0]*1001
parent = [i for i in range(1001)]

def find(x):
    if parent[x]==x:
        return x

    parent[x]=find(parent[x])
    return parent[x]

def union(x,y):
    a = find(x)
    b = find(y)

    if(a==b):
        return False
    elif(rank[a]>rank[b]):
        parent[b]=a
    elif(rank[b]>rank[a]):
        parent[a]=b
    else:
        parent[b]=a
        rank[a]+=1
    return True

inDegree = [0]*1001
```

```
        flag = 0
        # c is the reciever node..and a & b are the two nodes connecting to c
        for edge in edges:
            i,j = edge
            if(inDegree[j]>0):
                c = j
                a = inDegree[j]
                b = i
                flag = 1 #muliti inDegree exists


            else:
                inDegree[j] = i


        if flag!=1: #multi inDegree doesn't exist
            for edge in edges:
                if not union(*edge):
                    return edge
        else:
            edges.remove([b,c])
            for edge in edges:
                if not union(*edge):
                    return [a,c]  #if cycle exists, return the edge that's common in both cycle
and multi-inDegree
            return [b,c] #else return the normal multi edge node that was entered last
```