"Graph Valid Tree" (LeetCode #261, Medium)

Problem Description: Graph Valid Tree

Given `n` nodes labeled from `0` to `n-1` and a list of `edges`, determine if these edges form a valid tree.

Input:

- `n`: The number of nodes.
- `edges`: A 2D list where each element `[u, v]` represents an edge between nodes `u` and `v`.

Output:

- Return `True` if the graph is a valid tree; otherwise, return `False`

Example 1
Input: n = 5, edges = [[0,1],[0,2],[0,3],[1,4]]
Output: True

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

Key Observations

To determine if a graph is a valid tree:

1. **A tree must not have cycles**:
   - Use a cycle detection method.
2. **A tree must be connected**:
   - All nodes must be part of a single connected component.

Approach 1: Union-Find (Disjoint Set)

We can use the **Union-Find** data structure to check for:

1. **Cycles**:

- If two nodes are already connected and we try to add an edge between them, a cycle is formed.

2. **Connectivity**:
   - After processing all edges, there should be exactly one connected component (i.e., the number of edges must be n - 1

Algorithm

Steps:

1. Initialize a Union-Find structure with `parent` and `rank` arrays.
2. Iterate through each edge:
   - Use the `union` operation to connect nodes.
   - If `union` fails (i.e., nodes are already connected), a cycle is detected → return `False`.
3. After processing all edges, check:
   - The number of edges must be exactly n−1.
   - If not, the graph is disconnected → return `False`.
4. Return `True` if all checks pass.

Code:

```
class Solution:
    def validTree(self, n: int, edges: List[List[int]]) -> bool:
        # Step 1: Check basic condition (n - 1 edges for a valid tree)
        if len(edges) != n - 1:
            return False

        # Step 2: Initialize Union-Find
        parent = [i for i in range(n)]
        rank = [0] * n

        def find(x):
            if parent[x] != x:
                parent[x] = find(parent[x]) # Path compression
            return parent[x]

        def union(x, y):
```

```
            root_x = find(x)
            root_y = find(y)
            if root_x == root_y:
                return False # Cycle detected
            if rank[root_x] > rank[root_y]:
                parent[root_y] = root_x
            elif rank[root_x] < rank[root_y]:
                parent[root_x] = root_y
            else:
                parent[root_y] = root_x
                rank[root_x] += 1
            return True

        # Step 3: Process edges
        for u,v in edges:
            if not union(u, v):
                return False # Cycle detected

        # Step 4: Return True if no cycles and exactly n-1
edges
        return True
```

Time Complexity: for union-find operations: O(E. α(n)), where α(n) is the inverse Ackerman function(almost constant)
So Total TC: **O(E)**

Space Complexity:
O(n) for parent and rank arrays.