<https://leetcode.com/problems/number-of-good-paths/description/>

There is a tree (i.e. a connected, undirected graph with no cycles) consisting of n nodes numbered from 0 to n - 1 and exactly n - 1 edges.

You are given a **0-indexed** integer array vals of length n where vals[i] denotes the value of the ith node. You are also given a 2D integer array edges where edges[i] = [ai, bi] denotes that there exists an **undirected** edge connecting nodes ai and bi.

A **good path** is a simple path that satisfies the following conditions:

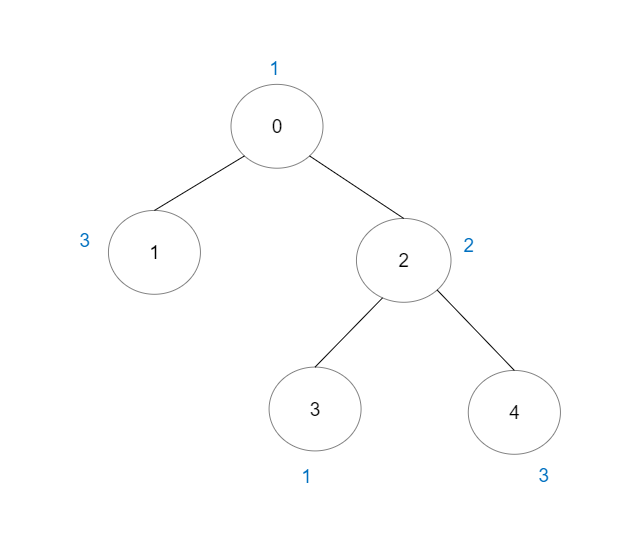
The starting node and the ending node have the **same** value.

All nodes between the starting node and the ending node have values **less than or equal to** the starting node (i.e. the starting node's value should be the maximum value along the path).

Return *the number of distinct good paths*.

Note that a path and its reverse are counted as the **same** path. For example, 0 -> 1 is considered to be the same as 1 -> 0. A single node is also considered as a valid path.

**Example 1:**



**Input:** vals = [1,3,2,1,3], edges = [[0,1],[0,2],[2,3],[2,4]]

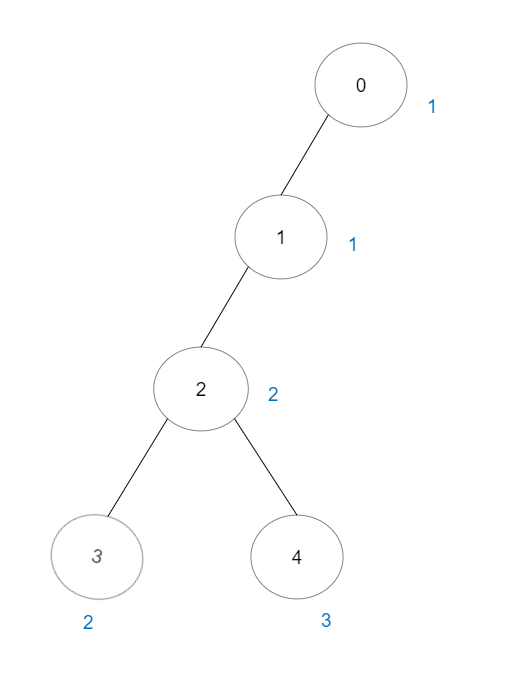
**Output:** 6

**Explanation:** There are 5 good paths consisting of a single node.

There is 1 additional good path: 1 -> 0 -> 2 -> 4.(The reverse path 4 -> 2 -> 0 -> 1 is treated as the same as 1 -> 0 -> 2 -> 4.)

Note that 0 -> 2 -> 3 is not a good path because vals[2] > vals[0].

**Example 2:**



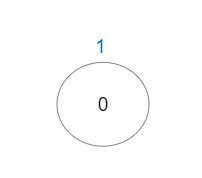
**Input:** vals = [1,1,2,2,3], edges = [[0,1],[1,2],[2,3],[2,4]]

**Output:** 7

**Explanation:** There are 5 good paths consisting of a single node.

There are 2 additional good paths: 0 -> 1 and 2 -> 3.

**Example 3:**



**Input:** vals = [1], edges = []

**Output:** 1

**Explanation:** The tree consists of only one node, so there is one good path.

**Constraints:**

n == vals.length

1 <= n <= 3 \* 10^4

0 <= vals[i] <= 10^5

edges.length == n - 1

edges[i].length == 2

0 <= ai, bi < n

ai != bi

edges represents a valid tree.

**Attempt 1: 2025-03-31**

**Solution 1: Hash Table + Union Find (180 min)**

class Solution {

    public int numberOfGoodPaths(int[] vals, int[][] edges) {

        int n = vals.length;

        int[] parent = new int[n];

        int[] rank = new int[n];

        for(int i = 0; i < n; i++) {

            parent[i] = i;

            rank[i] = 1;

        }

        // Build adjacency list

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

        for(int i = 0; i < n; i++) {

            adj.add(new ArrayList<>());

        }

        for(int[] edge : edges) {

            int u = edge[0];

            int v = edge[1];

            adj.get(u).add(v);

            adj.get(v).add(u);

        }

        // Group and sort nodes based on values in a TreeMap

        TreeMap<Integer, List<Integer>> valuesToNodes = new TreeMap<>();

        for(int i = 0; i < n; i++) {

            if(!valuesToNodes.containsKey(vals[i])) {

                valuesToNodes.put(vals[i], new ArrayList<>());

            }

            valuesToNodes.get(vals[i]).add(i);

        }

        // Each node is a good path by itself

        int result = n;

        for(int val : valuesToNodes.keySet()) {

            List<Integer> nodes = valuesToNodes.get(val);

            // Union adjacent nodes with values <= current val

            for(int u : nodes) {

                for(int v : adj.get(u)) {

                    if(vals[v] <= vals[u]) {

                        union(u, v, parent, rank);

                    }

                }

            }

            // Count nodes in each component

            Map<Integer, Integer> componentCount = new HashMap<>();

            // We only care about equal to root value's (u == v) node's count in

            // each group

            for(int u : nodes) {

                int root = find(u, parent);

                componentCount.put(root, componentCount.getOrDefault(root, 0) + 1);

            }

            // Calculate the number of good paths for current value

            for(int count : componentCount.values()) {

                result += count \* (count - 1) / 2;

            }

        }

        return result;

    }

    private int find(int x, int[] parent) {

        if(x == parent[x]) {

            return x;

        }

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

    }

    private void union(int x, int y, int[] parent, int[] rank) {

        int rootX = find(x, parent);

        int rootY = find(y, parent);

        if(rootX != rootY) {

            if(rank[rootX] > rank[rootY]) {

                parent[rootY] = rootX;

                rank[rootX] += rank[rootY];

            } else {

                parent[rootX] = rootY;

                rank[rootY] += rank[rootX];

            }

        }

    }

}

Time Complexity: O(nlogn)

Building adjacency list: O(n)

Sorting nodes by value: O(nlogn)

DSU operations (with path compression and union by rank):

O(α(n)) per operation, where α is the inverse Ackermann function (very slow-growing, effectively constant).

Total: O(nlogn) due to sorting.

Space Complexity: O(n)

**Refer to Deepseek**

This problem requires counting the number of "good" paths in a tree. A path is considered good if both endpoints have the same value, and all nodes between them have values less than or equal to that value.

**Approach**

Union-Find (Disjoint Set Union - DSU): We'll use this data structure to efficiently manage and merge sets of nodes.

Sort Nodes by Value: Process nodes in increasing order of their values.

Group Processing: For each group of nodes with the same value, count how many nodes in the current connected components can form good paths.

**Solution Code**

class Solution {

private int[] parent;

private int[] rank;

public int numberOfGoodPaths(int[] vals, int[][] edges) {

int n = vals.length;

parent = new int[n];

rank = new int[n];

for (int i = 0; i < n; i++) {

parent[i] = i;

rank[i] = 1;

}

// Build adjacency list

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

for (int i = 0; i < n; i++) {

adj.add(new ArrayList<>());

}

for (int[] edge : edges) {

int u = edge[0], v = edge[1];

adj.get(u).add(v);

adj.get(v).add(u);

}

// Map value to list of nodes

TreeMap<Integer, List<Integer>> valueToNodes = new TreeMap<>();

for (int i = 0; i < n; i++) {

valueToNodes.computeIfAbsent(vals[i], k -> new ArrayList<>()).add(i);

}

int result = n; // each node is a good path by itself

for (int val : valueToNodes.keySet()) {

List<Integer> nodes = valueToNodes.get(val);

// Union adjacent nodes with values <= current val

for (int u : nodes) {

for (int v : adj.get(u)) {

if (vals[v] <= vals[u]) {

union(u, v);

}

}

}

// Count nodes in each component

Map<Integer, Integer> componentCount = new HashMap<>();

for (int u : nodes) {

int root = find(u);

componentCount.put(root, componentCount.getOrDefault(root, 0) + 1);

}

// Calculate the number of good paths for current value

for (int count : componentCount.values()) {

result += count \* (count - 1) / 2;

}

}

return result;

}

private int find(int x) {

if (parent[x] != x) {

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

}

return parent[x];

}

private void union(int x, int y) {

int rootX = find(x);

int rootY = find(y);

if (rootX != rootY) {

if (rank[rootX] > rank[rootY]) {

parent[rootY] = rootX;

} else if (rank[rootX] < rank[rootY]) {

parent[rootX] = rootY;

} else {

parent[rootY] = rootX;

rank[rootX]++;

}

}

}

}

**Explanation**

Initialization: We initialize the DSU data structure with each node as its own parent.

Tree Construction: We build an adjacency list to represent the tree structure.

Value Grouping: Nodes are grouped by their values in a TreeMap to process them in increasing order.

Union Operations: For each node, we union it with its neighbors that have values less than or equal to its own value. This connects nodes that can potentially form good paths.

Counting Paths: For each group of nodes with the same value, we count how many nodes are in each connected component. The number of good paths formed by these nodes is calculated using the combination formula (n choose 2).

Result: The total number of good paths is the sum of all individual nodes (each is a path by itself) plus all combinations of nodes in the same component with the same value.

This approach efficiently counts all good paths by leveraging the union-find data structure and processing nodes in order of increasing values.

**Please elaborate in more detail how this solution works ? what's the logic behind ?**

**Problem Understanding**

We are given:

A tree with n nodes, where each node has a value vals[i].

A path between two nodes is called a good path if:

The starting and ending nodes have the same value.

All intermediate nodes in the path have values less than or equal to the starting/ending node's value.

Our goal is to count the number of such good paths in the tree.

**Key Observations**

**Tree Structure: Since it's a tree, there's exactly one unique path between any two nodes.**

**Good Path Conditions:**

**The path's endpoints must have the same value.**

**All nodes along the path must have values ≤ the endpoints' value.**

**Single Node Paths: Each individual node is trivially a good path (start and end at the same node).**

**Intuition**

**To form a good path between two nodes u and v with value val, all nodes on the path from u to v must have values ≤ val.**

**This implies that u and v must be connected through nodes with values ≤ val.**

**Therefore, we can think of merging (union) nodes with values ≤ val and then count how many nodes with value val are in each connected component.**

**Approach**

Union-Find (Disjoint Set Union - DSU):

This helps in efficiently managing and merging connected components.

Each node starts as its own parent.

The find operation finds the root parent of a node (with path compression for efficiency).

The union operation merges two sets.

Processing Nodes by Value:

We process nodes in increasing order of their values.

For each value val, we consider all nodes with value val.

We connect (union) these nodes with their neighbors that have values ≤ val.

After connecting, nodes with value val that are in the same connected component can form good paths among themselves.

Counting Good Paths:

For each value val, after processing, we count how many nodes with value val are in each connected component.

If a connected component has k nodes with value val, the number of good paths is k choose 2 (since any two nodes in the component can form a good path).

We add this to our result, which already includes the n single-node paths.

**Solution Code Breakdown**

Initialization:

parent and rank arrays for DSU.

Each node is its own parent initially.

Building Adjacency List:

Represents the tree structure for easy traversal.

Grouping Nodes by Value:

TreeMap<Integer, List<Integer>> valueToNodes groups nodes by their values in sorted order.

Processing Each Value:

For each value val in increasing order:

Get all nodes with value val.

Union each node with its neighbors having values ≤ val.

After union operations, count how many nodes with value val are in each connected component.

For each component with k nodes, add k choose 2 to the result.

DSU Operations:

find(x): Finds the root parent of x with path compression.

union(x, y): Merges the sets containing x and y based on rank.

**Example Walkthrough**

Consider the tree:

Nodes: 0 (1), 1 (3), 2 (2), 3 (1), 4 (2)

Edges: [[0,1],[1,2],[2,3],[2,4]]

Values: vals = [1,3,2,1,2]

Initialization:

All nodes are their own parents.

Processing Value 1:

Nodes: 0, 3

Union 0 with neighbors (1 has value 3 > 1 → skip).

Union 3 with neighbors (2 has value 2 > 1 → skip).

Components: {0}, {3}

Good paths: 2 (single nodes) + 0 (no pairs).

Processing Value 2:

Nodes: 2, 4

Union 2 with neighbors (1:3 > 2 → skip; 3:1 ≤ 2 → union; 4:2 ≤ 2 → union).

Union(2,3): parent[3] = 2

Union(2,4): parent[4] = 2

Components: {2,3,4}

Nodes with value 2 in component: 2,4 → count = 2

Good paths: 2 choose 2 = 1 (path 2-4)

Total so far: 5 (single nodes) + 1 = 6

Processing Value 3:

Node: 1

Union 1 with neighbors (0:1 ≤ 3 → union; 2:2 ≤ 3 → union).

Union(1,0): parent[0] = 1

Union(1,2): parent[2] = 1

Components: {0,1,2,3,4}

Nodes with value 3 in component: 1 → count = 1

Good paths: 1 choose 2 = 0

Total: 6 (no change)

Final result: 6 good paths.

**Why This Works**

By processing nodes in increasing order of values, we ensure that when we process value val, all nodes with values < val are already connected appropriately.

Nodes with value val can only form good paths among themselves if they're in the same component after connecting all nodes with values ≤ val.

The DSU efficiently manages these connections and component counts.

**Time Complexity**

Building adjacency list: O(n)

Sorting nodes by value: O(n log n)

DSU operations (with path compression and union by rank): O(α(n)) per operation, where α is the inverse Ackermann function (very slow-growing, effectively constant).

Total: O(n log n) due to sorting.

This approach efficiently counts all good paths by leveraging the union-find data structure and processing nodes in a strategic order.

**Why need "Value Grouping: Nodes are grouped by their values in a TreeMap to process them in increasing order." ?**

The key idea behind grouping nodes by their values and processing them in increasing order is to efficiently manage the connected components and ensure that we correctly count all possible good paths while maintaining the problem's constraints. Here's a detailed explanation:

**1. Understanding Good Paths**

A good path between two nodes u and v with value val requires:

vals[u] == vals[v] == val

All nodes on the path from u to v must have values ≤ val

This means that u and v must be connected through nodes that all have values ≤ val.

**2. Role of Union-Find (DSU)**

The Union-Find data structure helps us dynamically manage and merge connected components as we process nodes. Initially, each node is its own component. As we process nodes, we merge components based on the current value being processed.

**3. Why Process Nodes in Increasing Order of Values?**

**Processing nodes in increasing order of their values ensures that when we process a node with value val, all nodes with values < val have already been processed and merged appropriately. This is crucial because:**

**All relevant connections are established: For a node with value val, we only care about connections through nodes with values ≤ val. By processing in increasing order, we guarantee that all such connections are already considered when we process val.**

**Avoids missing valid paths: If we processed nodes in arbitrary order, we might miss connections that could form good paths because some necessary merges (unions) might not have happened yet.**

**4. Value Grouping via TreeMap**

TreeMap: This data structure automatically sorts keys (node values) in ascending order. By using TreeMap, we can easily process nodes from the smallest value to the largest.

Grouping Nodes by Value: For each value val, we get all nodes that have this value. This allows us to:

Union these nodes with their neighbors: We connect nodes with value val to their adjacent nodes that have values ≤ val. This ensures that nodes with value val are in the correct connected components.

Count good paths: After processing all connections for value val, we count how many nodes with value val are in each connected component. The number of good paths for these nodes is the number of ways to choose 2 nodes from the same component, which is k choose 2 for k nodes.

**5. Example to Illustrate**

Consider the tree:

Nodes: 0 (1), 1 (3), 2 (2), 3 (1), 4 (2)

Edges: [[0,1],[1,2],[2,3],[2,4]]

Values: vals = [1,3,2,1,2]

Processing Order: Values in order: 1, 2, 3.

Value 1: Nodes 0 and 3.

No neighbors with values ≤ 1, so no unions.

Components: {0}, {3}.

Good paths: 2 (single nodes).

Value 2: Nodes 2 and 4.

Union 2 with neighbors (3:1 ≤ 2, 4:2 ≤ 2).

Union(2,3): parent[3] = 2.

Union(2,4): parent[4] = 2.

Components: {2,3,4}.

Nodes with value 2: 2,4 → 2 choose 2 = 1 good path (2-4).

Value 3: Node 1.

Union 1 with neighbors (0:1 ≤ 3, 2:2 ≤ 3).

Union(1,0): parent[0] = 1.

Union(1,2): parent[2] = 1.

Components: {0,1,2,3,4}.

Nodes with value 3: 1 → no pairs.

Total Good Paths: 2 (single) + 1 (2-4) = 3.

If we didn't process in increasing order, we might miss some unions. For example, if we processed value 2 before value 1, node 3 (value 1) wouldn't be connected to node 2 (value 2), leading to incorrect counts.

**6. Why Not Process in Decreasing Order?**

Processing in decreasing order would violate the condition that all intermediate nodes must have values ≤ val. For a node with value val, we need all nodes with values ≤ val to be connected before counting good paths. If we process larger values first, we might incorrectly include paths where intermediate nodes have higher values, violating the problem's constraints.

**7. Efficiency**

TreeMap Sorting: O(n log n) for sorting nodes by value.

Union-Find Operations: Nearly O(1) per operation with path compression and union by rank.

Overall: O(n log n) due to sorting, which is efficient for the problem constraints.

**Conclusion**

Grouping nodes by their values and processing them in increasing order is essential to:

Ensure that all necessary connections (unions) are made before counting good paths.

Maintain the invariant that all intermediate nodes in a good path have values ≤ the endpoint values.

Efficiently and correctly count all possible good paths using the Union-Find data structure.

This approach guarantees that we correctly account for all valid paths while adhering to the problem's constraints, making it both optimal and intuitive.

**Refer to**

[Union Find (并查集) 的四种方法](note://9E9311B4A1DC4AA68AB4E855322B9ADB)

[L323.Lint431.Number of Connected Components in an Undirected Graph (Ref.L2421)](note://EFD0063AEF404780A207CA32B7D63DB2)

[L547.Number of Provinces (Friend Circles) (Ref.L323,L2421)](note://55FAE6CE6252461BA88F8CDE05E94F2E)

[L1671.Minimum Number of Removals to Make Mountain Array (Ref.L300)](note://A717168457D646DF8CE8A9054DD61C12)

[L2506.Count Pairs Of Similar Strings (Ref.L451)](note://WEB3e9b6b9eba3c0e5a162c03afb590c70b)