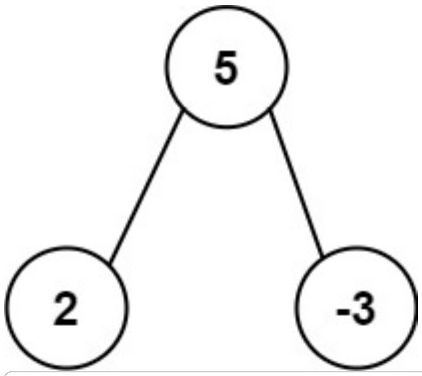
508. Most Frequent Subtree Sum

Given the `root` of a binary tree, return the most frequent **subtree sum**. If there is a tie, return all the values with the highest frequency in any order.

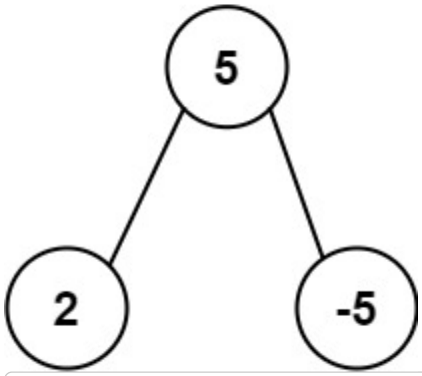
The **subtree sum** of a node is defined as the sum of all the node values formed by the subtree rooted at that node (including the node itself).

Example 1:



```
Input: root = [5,2,-3]
Output: [2,-3,4]
```

Example 2:



```
Input: root = [5,2,-5]
Output: [2]
```

Constraints:

- The number of nodes in the tree is in the range `[1, 10^4]`.
- `-10^5 <= Node.val <= 10^5`

```
def findFrequentTreeSum(self, root: Optional[TreeNode]) -> List[int]:
       mapp = defaultdict(int)
    ans= []
    def dfs(root):
        nonlocal mapp
        if not root:return 0

        val = root.val+dfs(root.left)+dfs(root.right)

        mapp[val]+=1
        return val

    dfs(root)
    maxxval = max(mapp.values())
    for i,j in mapp.items():
        if maxxval == j:
            ans.append(i)
    return ans
```

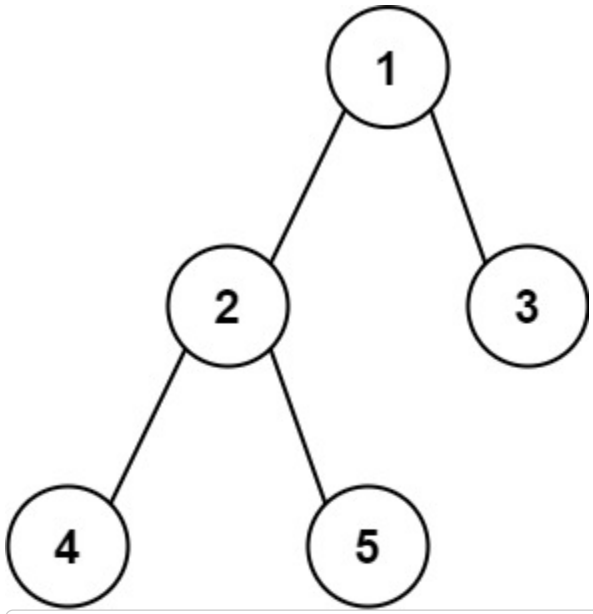
543. Diameter of Binary Tree

Given the `root` of a binary tree, return *the length of the* ***diameter*** *of the tree*.

The **diameter** of a binary tree is the **length** of the longest path between any two nodes in a tree. This path may or may not pass through the `root`.

The **length** of a path between two nodes is represented by the number of edges between them.

Example 1:



```
Input: root = [1,2,3,4,5]
Output: 3
Explanation: 3 is the length of the path [4,2,1,3] or [5,2,1,3].
```

Example 2:

```
Input: root = [1,2]
Output: 1
```

Constraints:

- The number of nodes in the tree is in the range `[1, 10`^{4}`]`.
- `-100 <= Node.val <= 100`

Diameter of binary tree - Important

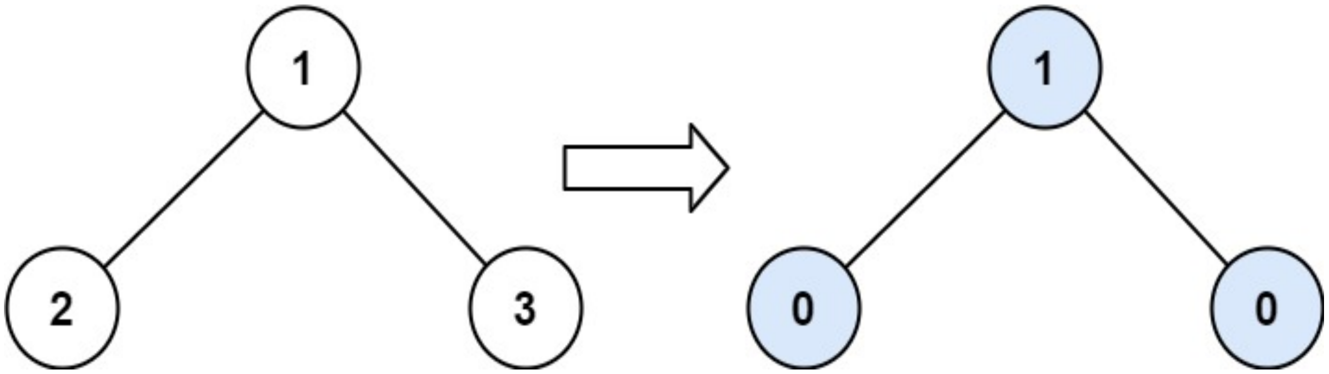
```
import sys
class Solution:
    def diameterOfBinaryTree(self, root: Optional[TreeNode]) -> int:
        def solve(root):
            nonlocal res
            if root == None:
                return 0
            left=solve(root.left)
            right=solve(root.right)

            temp=max(left,right)+1
            ans=max(temp,left+right+1)
            res=max(res,ans)
            return temp

        res=-float("inf")
        solve(root)
        return res-1
```

563. Binary Tree Tilt

Given the `root` of a binary tree, return *the sum of every tree node's* ***tilt***.

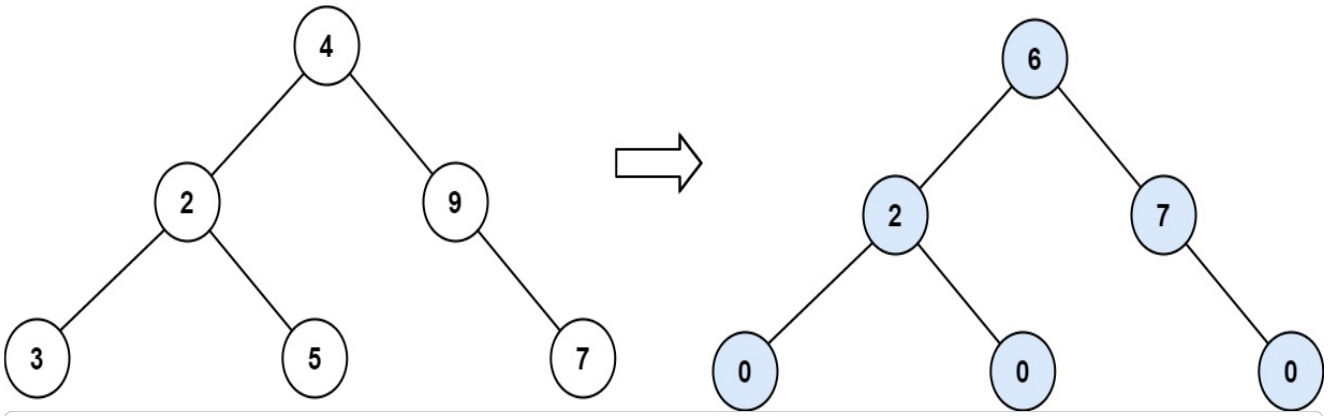
The **tilt** of a tree node is the **absolute difference** between the sum of all left subtree node **values** and all right subtree node **values**. If a node does not have a left child, then the sum of the left subtree node **values** is treated as `0`. The rule is similar if the node does not have a right child.

Example 1:



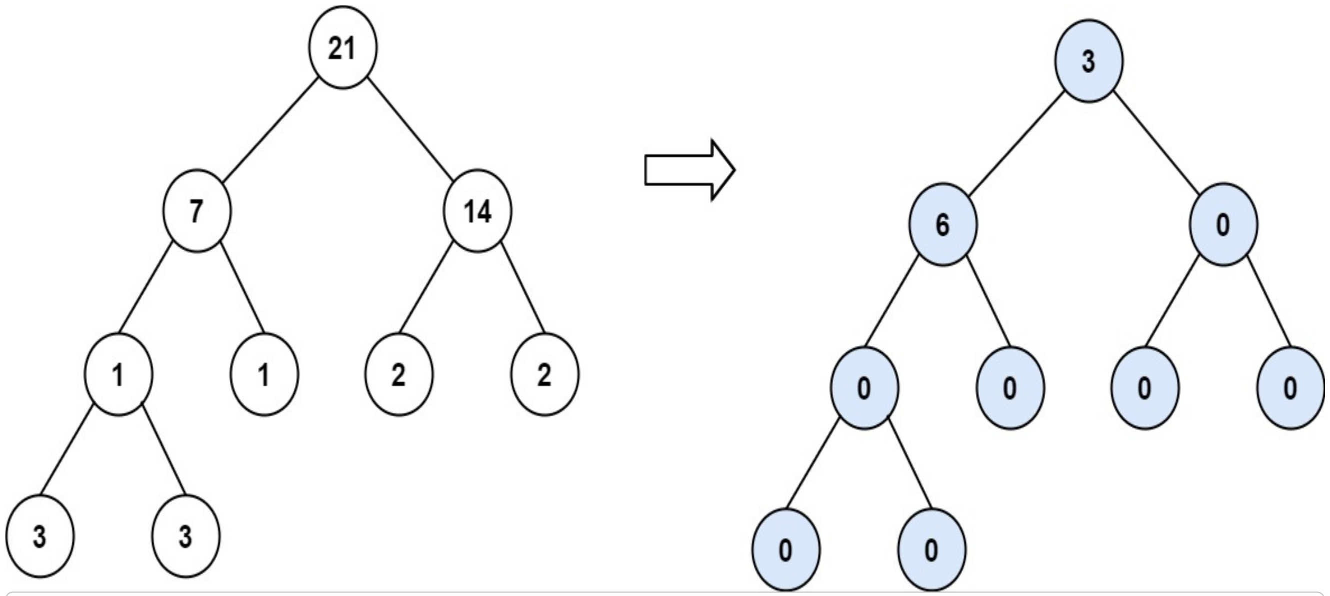
```
Input: root = [1,2,3]
Output: 1
Explanation:
Tilt of node 2 : |0-0| = 0 (no children)
Tilt of node 3 : |0-0| = 0 (no children)
Tilt of node 1 : |2-3| = 1 (left subtree is just left child, so sum is 2; right subtr
Sum of every tilt : 0 + 0 + 1 = 1
```

Example 2:



```
Input: root = [4,2,9,3,5,null,7]
Output: 15
Explanation:
Tilt of node 3 : |0-0| = 0 (no children)
Tilt of node 5 : |0-0| = 0 (no children)
Tilt of node 7 : |0-0| = 0 (no children)
Tilt of node 2 : |3-5| = 2 (left subtree is just left child, so sum is 3; right subtr
Tilt of node 9 : |0-7| = 7 (no left child, so sum is 0; right subtree is just right c
Tilt of node 4 : |(3+5+2)-(9+7)| = |10-16| = 6 (left subtree values are 3, 5, and 2,
Sum of every tilt : 0 + 0 + 0 + 2 + 7 + 6 = 15
```

Example 3:



```
Input: root = [21,7,14,1,1,2,2,3,3]
Output: 9
```

Constraints:

- The number of nodes in the tree is in the range `[0, 10^4]`.
- `-1000 <= Node.val <= 1000`

```
def findTilt(self, root: Optional[TreeNode]) -> int:
        ans = 0
        if not root:return 0
        def dfs(node):
                nonlocal ans
                L = 0
                R = 0
                if node.left:
                        L = dfs(node.left)
                if node.right:
                        R = dfs(node.right)
                if not node.right and not node.left:return node.val
                else:ans += abs(L-R)
                return L+R+ node.val
        dfs(root)
        return ans
```

662. Maximum Width of Binary Tree

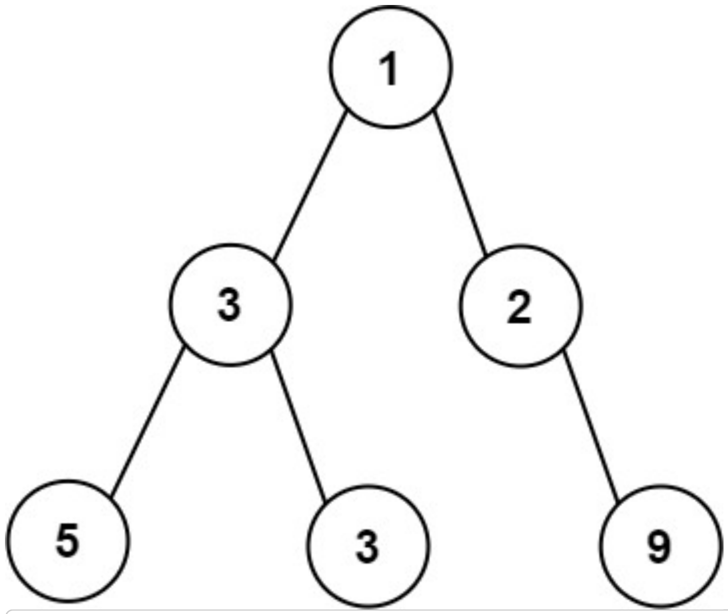
Given the `root` of a binary tree, return *the* ***maximum width*** *of the given tree*.

The **maximum width** of a tree is the maximum **width** among all levels.

The **width** of one level is defined as the length between the end-nodes (the leftmost and rightmost non-null nodes), where the null nodes between the end-nodes that would be present in a complete binary tree extending down to that level are also counted into the length calculation.

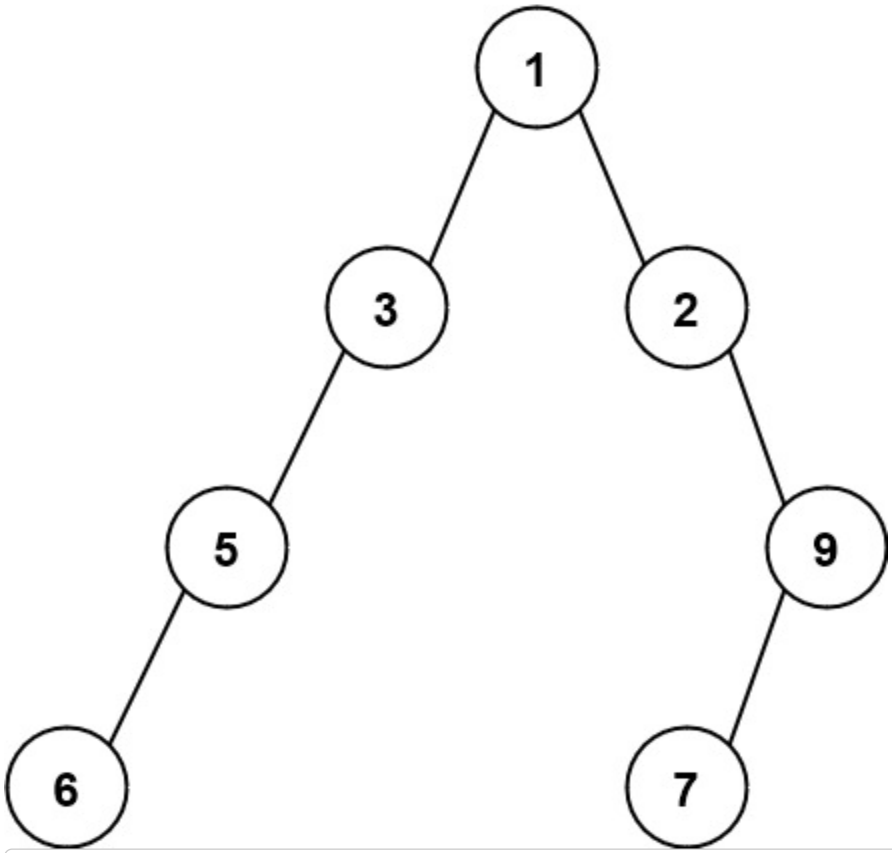
It is **guaranteed** that the answer will in the range of a **32-bit** signed integer.

Example 1:



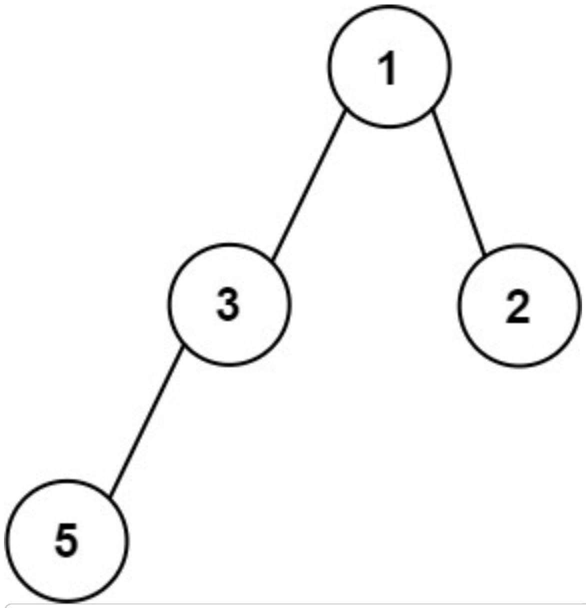
```
Input: root = [1,3,2,5,3,null,9]
Output: 4
Explanation: The maximum width exists in the third level with length 4 (5,3,null,9).
```

Example 2:



```
Input: root = [1,3,2,5,null,null,9,6,null,7]
Output: 7
Explanation: The maximum width exists in the fourth level with length 7 (6,null,null,
```

Example 3:



```
Input: root = [1,3,2,5]
Output: 2
Explanation: The maximum width exists in the second level with length 2 (3,2).
```

Constraints:

- The number of nodes in the tree is in the range `[1, 3000]`.
- `-100 <= Node.val <= 100`

Concepts to be known: For a binary tree index of left child of a node -> 2 * parent node level + 1 index of right child of a node -> 2 * parent node level + 2 width = right_most node index - left_most node index + 1 (for a single level) Approach One queue for keeping track for all nodes One list (named: nodes) to store all nodes' indexes for a level Ans = max(Ans , current level's width)

```
class Solution:
    def widthOfBinaryTree(self, root: Optional[TreeNode]) -> int:
        queue = [(0,root)]
        ans = 0
        while queue:
            n =len(queue)
            nodes = []
            for _ in range(n):
                idx,node = queue.pop(0)
                nodes.append(idx)
                if node.left:
                    queue.append((2*idx+1,node.left))
                if node.right:
                    queue.append((2*idx+2,node.right))
            ans = max(ans,max(nodes)-min(nodes)+1)
        return ans
```

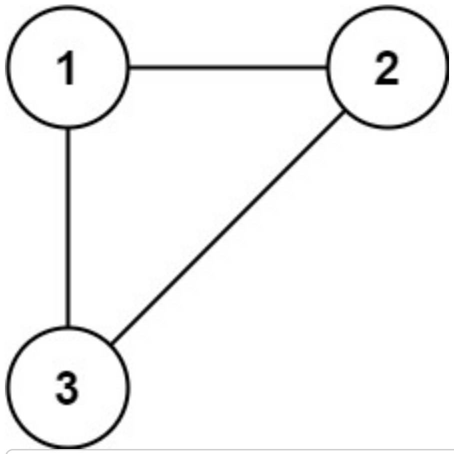
684. Redundant Connection

In 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] = [`a_i`, `b_i`]` indicates that there is an edge between nodes a_i and b_i in the graph.

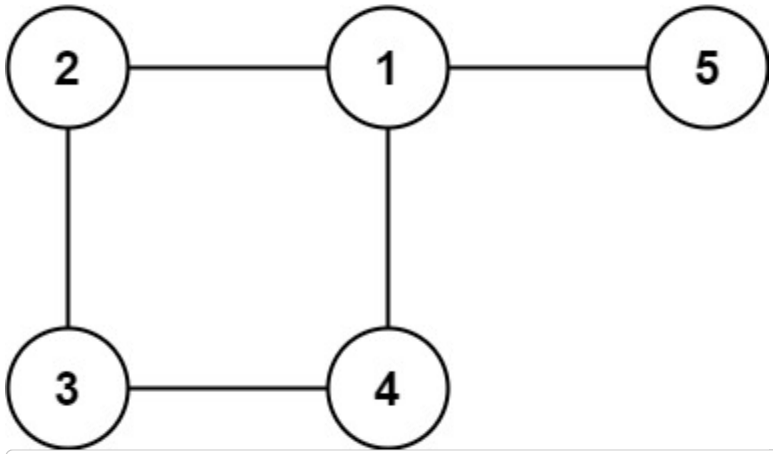
Return *an edge that can be removed so that the resulting graph is a tree of* `n` *nodes*. If there are multiple answers, return the answer that occurs last in the input.

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 <=` a_i `<` b_i `<= edges.length`
- a_i `!=` b_i
- There are no repeated edges.
- The given graph is connected.

Important : UnionFind

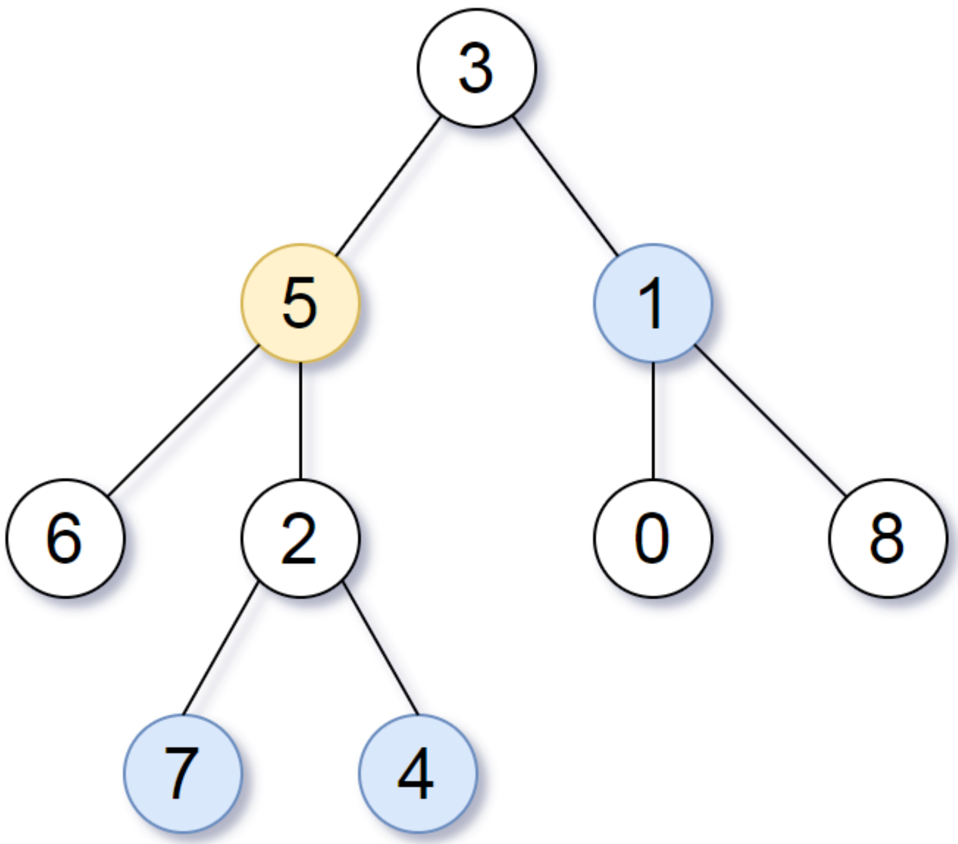
```
class Solution:
    def findRedundantConnection(self, edges: List[List[int]]) -> List[int]:
        parent = [i for i in range(len(edges) + 1)]  # Parent array to track disjoi
nt sets
        rank = [1] * (len(edges) + 1)  # Rank array for union by rank optimization

        # Find function with path compression
        def find(n):
            while n != parent[n]:
                parent[n] = parent[parent[n]]  # Path compression
                n = parent[n]
            return n

        # Union function with union by rank
        def union(n1, n2):
            par1, par2 = find(n1), find(n2)
            if par1 == par2:  # If they already share the same parent, this edge is
redundant
                return False
            # Union by rank
            if rank[par1] > rank[par2]:
                parent[par2] = par1
                rank[par1] += rank[par2]
            else:
                parent[par1] = par2
                rank[par2] += rank[par1]
            return True

        # Process each edge
        for i, j in edges:
            if not union(i, j):  # If union fails, the edge is redundant
                return [i, j]
```

863. All Nodes Distance K in Binary Tree

Given the `root` of a binary tree, the value of a target node `target`, and an integer `k`, return *an array of the values of all nodes that have a distance* `k` *from the target node.*

You can return the answer in **any order**.

Example 1:



```
Input: root = [3,5,1,6,2,0,8,null,null,7,4], target = 5, k = 2
Output: [7,4,1]
Explanation: The nodes that are a distance 2 from the target node (with value 5) have
```

Example 2:

```
Input: root = [1], target = 1, k = 3
Output: []
```

Constraints:

- The number of nodes in the tree is in the range `[1, 500]`.
- `0 <= Node.val <= 500`
- All the values `Node.val` are **unique**.
- `target` is the value of one of the nodes in the tree.
- `0 <= k <= 1000`

Very Important

```
# Definition for a binary tree node.

class Solution:
    def distanceK(self, node: TreeNode, target: TreeNode, k: int) -> List[int]:
        mapp = defaultdict(int)
        def preorder(root,parent):
            nonlocal mapp
            if not root: return
            mapp[root] = parent
            preorder(root.left,root)
            preorder(root.right,root)

        preorder(node,None)
        queue = [target]
        visited = set()
        level = 0
        while queue:
            n =len(queue)
            if level == k:
                break
            level+=1
            for _ in range(n):
                Node = queue.pop(0)
                #print(Node.val)
                visited.add(Node)
                if mapp[Node] != None and mapp[Node] not in visited:
                    queue.append(mapp[Node])
                if Node.left and Node.left not in visited:
                    queue.append(Node.left)
                if Node.right and Node.right not in visited:
                    queue.append(Node.right)
        for i in range(len(queue)):
            queue[i] = queue[i].val
        return queue
```

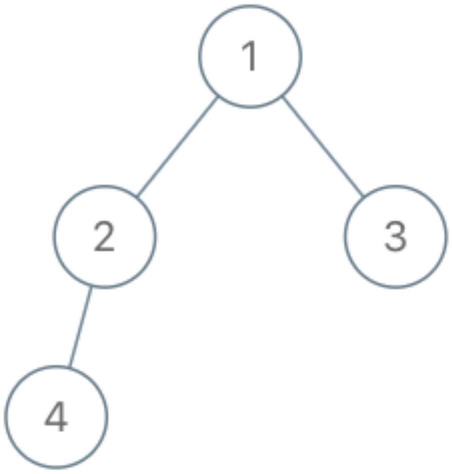
993. Cousins in Binary Tree

Given the `root` of a binary tree with unique values and the values of two different nodes of the tree `x` and `y`, return `true` *if the nodes corresponding to the values* `x` *and* `y` *in the tree are* ***cousins****, or* `false` *otherwise.*

Two nodes of a binary tree are **cousins** if they have the same depth with different parents.

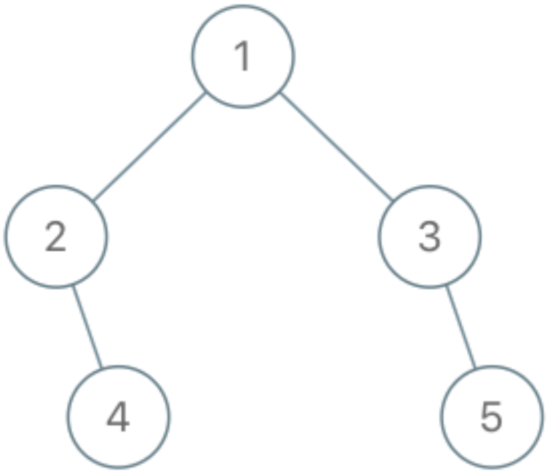
Note that in a binary tree, the root node is at the depth `0`, and children of each depth `k` node are at the depth `k + 1`.

Example 1:



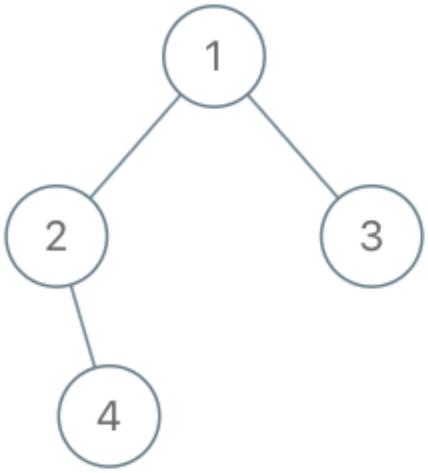
```
Input: root = [1,2,3,4], x = 4, y = 3
Output: false
```

Example 2:



```
Input: root = [1,2,3,null,4,null,5], x = 5, y = 4
Output: true
```

Example 3:



```
Input: root = [1,2,3,null,4], x = 2, y = 3
Output: false
```

Constraints:

- The number of nodes in the tree is in the range `[2, 100]`.
- `1 <= Node.val <= 100`
- Each node has a **unique** value.
- `x != y`
- `x` and `y` are exist in the tree.

```
class Solution:
    def isCousins(self, root: Optional[TreeNode], x: int, y: int) -> bool:
        queue = [(0,root)]
        ans= []
        level = 0
        while queue:
            level+=1
            n = len(queue)
            for _ in range(n):
                idx,node = queue.pop(0)
                if node.val in [x,y]:
                    ans.append([idx,level])
                    if len(ans)==2:
                        break
                if node.left:
                    queue.append([2*idx+1,node.left])
                if node.right:
                    queue.append([2*idx+2,node.right])
        if ans[0][1] == ans[1][1]:
            idx1 =(ans[0][0]-1)//2
            idx2 =(ans[1][0]-1)//2
            return True if idx2 != idx1 else False
        return False
```

998. Maximum Binary Tree II

A **maximum tree** is a tree where every node has a value greater than any other value in its subtree.

You are given the `root` of a maximum binary tree and an integer `val`.

Just as in the previous problem (https://leetcode.com/problems/maximum-binary-tree/), the given tree was constructed from a list `a` (`root = Construct(a)`) recursively with the following `Construct(a)` routine:

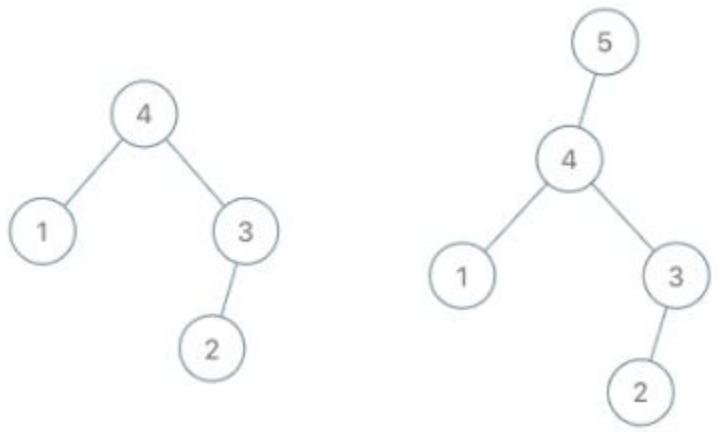
- If `a` is empty, return `null`.
- Otherwise, let `a[i]` be the largest element of `a`. Create a `root` node with the value `a[i]`.
- The left child of `root` will be `Construct([a[0], a[1], ..., a[i - 1]])`.
- The right child of `root` will be `Construct([a[i + 1], a[i + 2], ..., a[a.length - 1]])`.
- Return `root`.

Note that we were not given `a` directly, only a root node `root = Construct(a)`.

Suppose `b` is a copy of `a` with the value `val` appended to it. It is guaranteed that `b` has unique values.

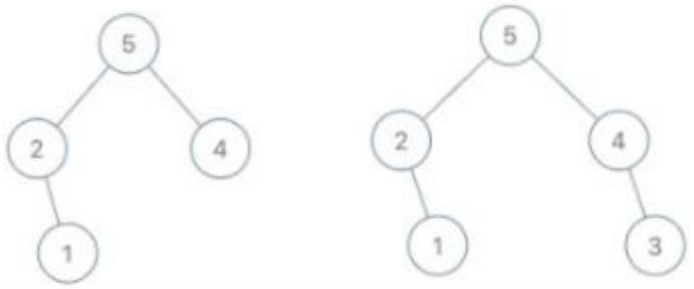
Return `Construct(b)`.

Example 1:



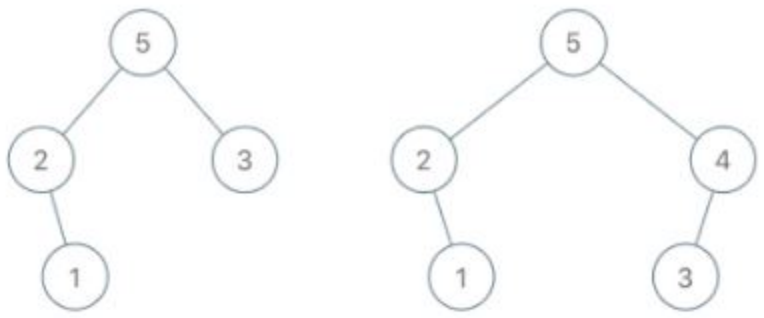
```
Input: root = [4,1,3,null,null,2], val = 5
Output: [5,4,null,1,3,null,null,2]
Explanation: a = [1,4,2,3], b = [1,4,2,3,5]
```

Example 2:



```
Input: root = [5,2,4,null,1], val = 3
Output: [5,2,4,null,1,null,3]
Explanation: a = [2,1,5,4], b = [2,1,5,4,3]
```

Example 3:



```
Input: root = [5,2,3,null,1], val = 4
Output: [5,2,4,null,1,3]
Explanation: a = [2,1,5,3], b = [2,1,5,3,4]
```

Constraints:

- The number of nodes in the tree is in the range `[1, 100]`.
- `1 <= Node.val <= 100`
- All the values of the tree are **unique**.
- `1 <= val <= 100`

Maximum binary tree

```
class Solution:
    def insertIntoMaxTree(self, root: Optional[TreeNode], val: int) -> Optional[Tre
eNode]:
        def dfs(root):
            temp = [root.val]
            if root.left:
                temp = dfs(root.left) + temp
            if root.right:
                temp += dfs(root.right)
            return temp
        newarr = dfs(root)
        newarr.append(val)
        def construct(nums):
            if not nums:return None
            root = max(nums)
            root_ind = nums.index(root)
            node = TreeNode(root)
            node.left = construct(nums[:root_ind])
            node.right = construct(nums[root_ind+1:])
            return node
        return construct(newarr)
```

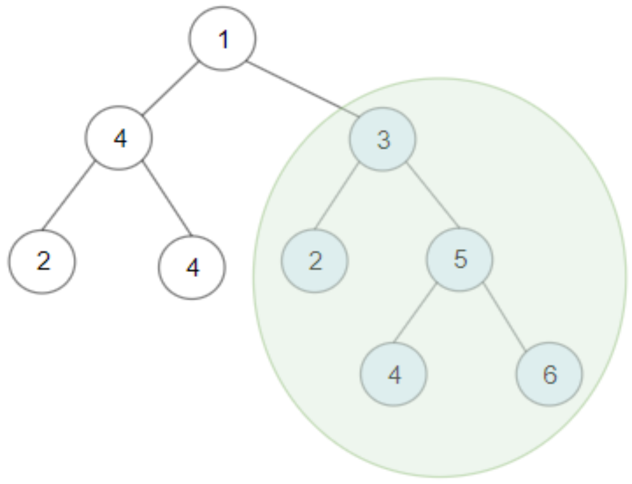
1373. Maximum Sum BST in Binary Tree

Given a **binary tree** `root`, return *the maximum sum of all keys of **any** sub-tree which is also a Binary Search Tree (BST).*

Assume a BST is defined as follows:

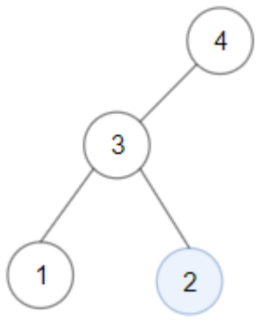
- The left subtree of a node contains only nodes with keys **less than** the node's key.
- The right subtree of a node contains only nodes with keys **greater than** the node's key.
- Both the left and right subtrees must also be binary search trees.

Example 1:



```
Input: root = [1,4,3,2,4,2,5,null,null,null,null,null,null,4,6]
Output: 20
Explanation: Maximum sum in a valid Binary search tree is obtained in root node with
```

Example 2:



```
Input: root = [4,3,null,1,2]
Output: 2
Explanation: Maximum sum in a valid Binary search tree is obtained in a single root r
```

Example 3:

```
Input: root = [-4,-2,-5]
Output: 0
Explanation: All values are negatives. Return an empty BST.
```

Constraints:

- The number of nodes in the tree is in the range `[1, 4 * 10^4]`.
- `-4 * 10^4 <= Node.val <= 4 * 10^4`

```
#Important Hard
    def maxSumBST(self, root):
    max_sum = 0
    def kunction(node):
        nonlocal max_sum
        if node is None:
            return 0, float("inf"), float("-inf")

        l_sum, l_min, l_max = kunction(node.left)
        r_sum, r_min, r_max = kunction(node.right)

        if l_max < node.val < r_min:
            max_sum = max(max_sum,node.val+l_sum+r_sum)
            return node.val+l_sum+r_sum, min(l_min,node.val), max(r_max,node.val)

        return 0, float("-inf"), float("inf")

    kunction(root)

    return max_sum
```

1376. Time Needed to Inform All Employees

A company has `n` employees with a unique ID for each employee from `0` to `n - 1`. The head of the company is the one with `headID`.

Each employee has one direct manager given in the `manager` array where `manager[i]` is the direct manager of the `i-th` employee, `manager[headID] = -1`. Also, it is guaranteed that the subordination relationships have a tree structure.

The head of the company wants to inform all the company employees of an urgent piece of news. He will inform his direct subordinates, and they will inform their subordinates, and so on until all employees know about the urgent news.

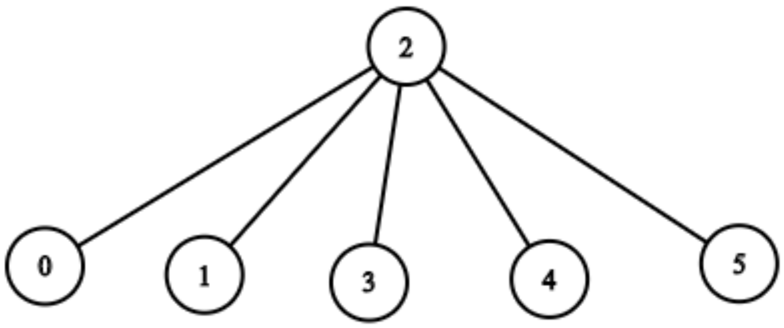
The `i-th` employee needs `informTime[i]` minutes to inform all of his direct subordinates (i.e., After informTime[i] minutes, all his direct subordinates can start spreading the news).

Return *the number of minutes* needed to inform all the employees about the urgent news.

Example 1:

```
Input: n = 1, headID = 0, manager = [-1], informTime = [0]
Output: 0
Explanation: The head of the company is the only employee in the company.
```

Example 2:



```
Input: n = 6, headID = 2, manager = [2,2,-1,2,2,2], informTime = [0,0,1,0,0,0]
Output: 1
Explanation: The head of the company with id = 2 is the direct manager of all the emp
The tree structure of the employees in the company is shown.
```

Constraints:

- `1 <= n <= 10`^{5}
- `0 <= headID < n`
- `manager.length == n`
- `0 <= manager[i] < n`
- `manager[headID] == -1`
- `informTime.length == n`
- `0 <= informTime[i] <= 1000`
- `informTime[i] == 0` if employee `i` has no subordinates.
- It is **guaranteed** that all the employees can be informed.

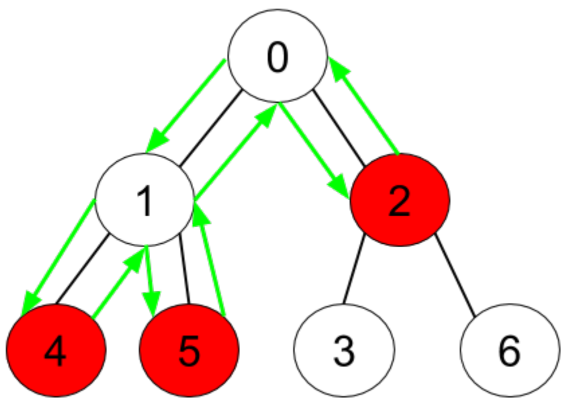
```
class Solution:
    def numOfMinutes(self, n: int, headID: int, manager: List[int], informTime: Lis
t[int]) -> int:
        ans = 0
        mapp = defaultdict(list)
        for i,j in enumerate(manager):
            mapp[j].append(i)
        def dfs(node):
            if node not in mapp:
                return 0
            max_time = 0
            for i in mapp[node]:
                max_time = max(max_time,informTime[node] + dfs(i))
            return max_time
        return dfs(headID)
```

1443. Minimum Time to Collect All Apples in a Tree

Given an undirected tree consisting of `n` vertices numbered from `0` to `n-1`, which has some apples in their vertices. You spend 1 second to walk over one edge of the tree. *Return the minimum time in seconds you have to spend to collect all apples in the tree, starting at **vertex 0** and coming back to this vertex.*

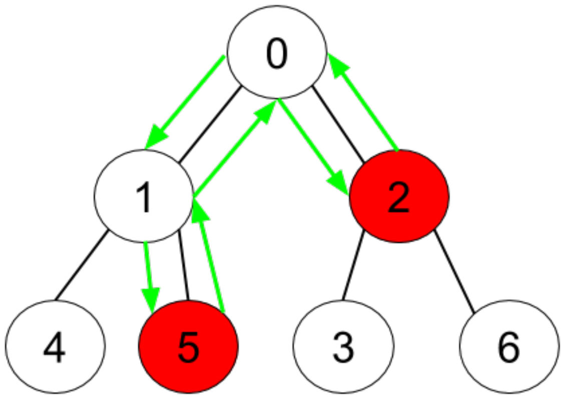
The edges of the undirected tree are given in the array `edges`, where `edges[i] = [a_i, b_i]` means that exists an edge connecting the vertices `a_i` and `b_i`. Additionally, there is a boolean array `hasApple`, where `hasApple[i] = true` means that vertex `i` has an apple; otherwise, it does not have any apple.

Example 1:



```
Input: n = 7, edges = [[0,1],[0,2],[1,4],[1,5],[2,3],[2,6]], hasApple = [false,false,
Output: 8
Explanation: The figure above represents the given tree where red vertices have an ap
```

Example 2:



```
Input: n = 7, edges = [[0,1],[0,2],[1,4],[1,5],[2,3],[2,6]], hasApple = [false,false,
Output: 6
Explanation: The figure above represents the given tree where red vertices have an ap
```

Example 3:

```
Input: n = 7, edges = [[0,1],[0,2],[1,4],[1,5],[2,3],[2,6]], hasApple = [false,false,
Output: 0
```

Constraints:

- `1 <= n <= 10`^{5}
- `edges.length == n - 1`
- `edges[i].length == 2`
- `0 <=` a_i `<` b_i `<= n - 1`
- `hasApple.length == n`

```
    def minTime(self, n: int, edges: List[List[int]], hasApple: List[bool]) -> int:
        mapp = {i:[] for i in range(n)}
        for k,v in edges:
            mapp[k].append(v)
            mapp[v].append(k)
        ans = 0
        visited = set()
        def dfs(node):
            nonlocal visited,mapp,ans
            have_child = False
            visited.add(node)
            for i in mapp[node]:
                if i not in visited:
                    have_child = dfs(i) or have_child
            if hasApple[node] or have_child:
                ans += 2
            return hasApple[node] or have_child
        dfs(0)
        return ans if ans == 0 else ans-2
```

2421. Number of Good Paths

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 i^{th} node. You are also given a 2D integer array `edges` where `edges[i] = [`a_i`, `b_i`]` denotes that there exists an **undirected** edge connecting nodes a_i and b_i.

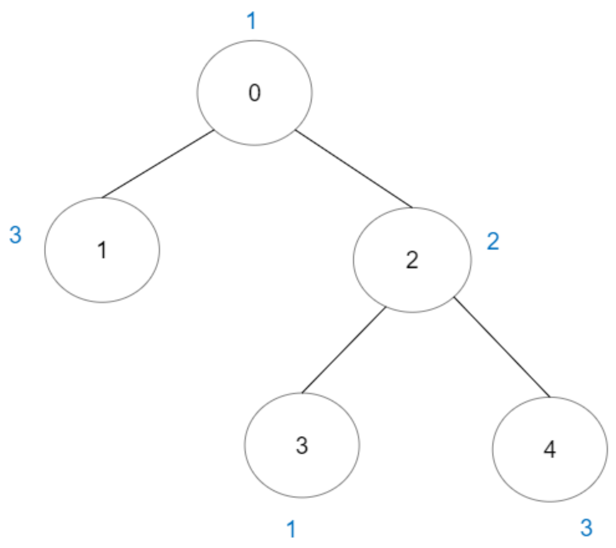
A **good path** is a simple path that satisfies the following conditions:

1. The starting node and the ending node have the **same** value.
2. 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*.

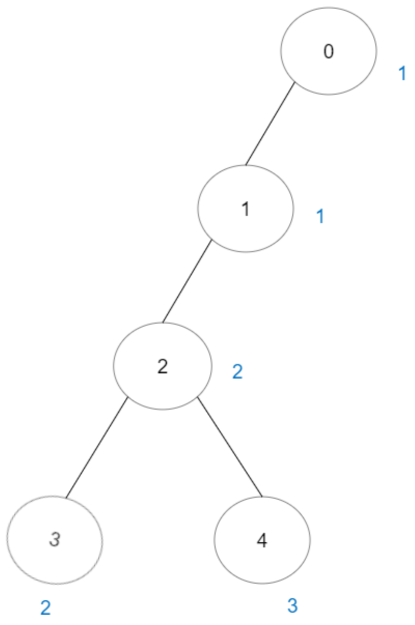
Note that a path and its reverse are counted as the **same** path. For example, `0 -> 1` is considered to be the same as `1 -> 0`. A single node is also considered as a valid path.

Example 1:



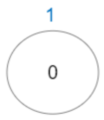
```
Input: vals = [1,3,2,1,3], edges = [[0,1],[0,2],[2,3],[2,4]]
Output: 6
Explanation: There are 5 good paths consisting of a single node.
There is 1 additional good path: 1 -> 0 -> 2 -> 4.
(The reverse path 4 -> 2 -> 0 -> 1 is treated as the same as 1 -> 0 -> 2 -> 4.)
Note that 0 -> 2 -> 3 is not a 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 <=` a_i, b_i `< n`
- a_i `!=` b_i
- `edges` represents a valid tree.

UnionFind

```
class UnionFind:
    def __init__(self,n):
        self.par = list(range(n))
        self.rank = [0]*n

    def find(self,n):
        while n != self.par[n]:
            self.par[n] = self.par[self.par[n]]
            n = self.par[n]
        return n

    def union(self, a, b):
        aRoot = self.find(a)
        bRoot = self.find(b)
        if aRoot == bRoot:
            return False
        elif self.rank[aRoot] < self.rank[bRoot]:
            self.rank[bRoot] += self.rank[aRoot]
            self.par[aRoot] = bRoot
        else:
            self.rank[aRoot] += self.rank[bRoot]
            self.par[bRoot] = aRoot
        return True

class Solution:
    def numberOfGoodPaths(self, vals: List[int], edges: List[List[int]]) -> int:
        adj = defaultdict(list)
        for a,b in edges:
            adj[a].append(b)
            adj[b].append(a)

        valToIndex = defaultdict(list)
        for ind,val in enumerate(vals):
            valToIndex[val].append(ind)

        res = 0
        uf = UnionFind(len(vals))
        for val in sorted(valToIndex.keys()):
            for node in valToIndex[val]:
                for nei in adj[node]:
                    if vals[nei] <= vals[node]:
                        uf.union(nei,node)

            count = defaultdict(int)
            for i in valToIndex[val]:
                root =  uf.find(i)
                count[root] += 1
                res += count[root]
        return res
```

2471. Minimum Number of Operations to Sort a Binary Tree by Level

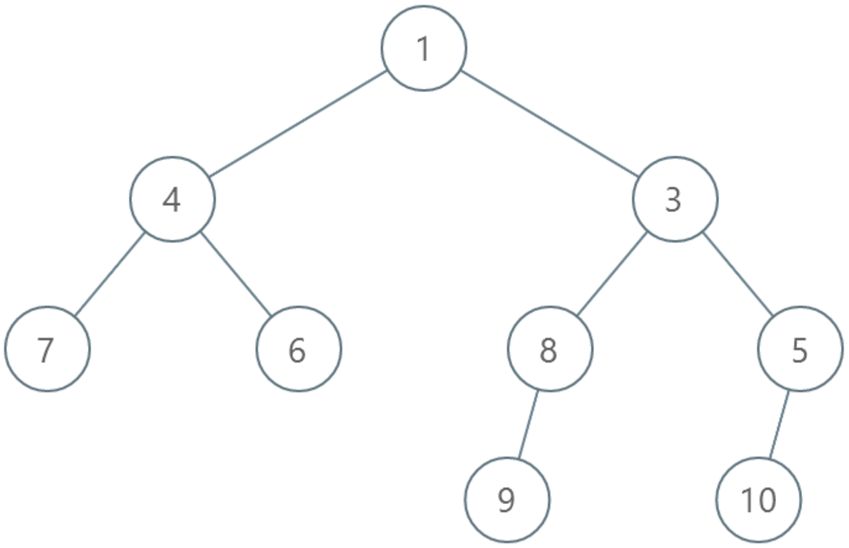
You are given the `root` of a binary tree with **unique values**.

In one operation, you can choose any two nodes **at the same level** and swap their values.

Return *the minimum number of operations needed to make the values at each level sorted in a **strictly increasing order***.

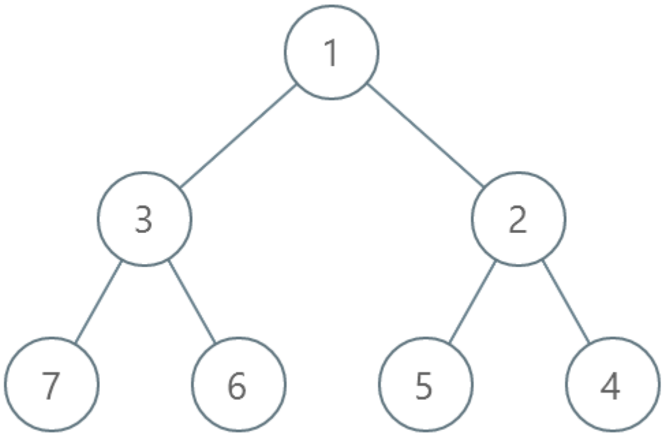
The **level** of a node is the number of edges along the path between it and the root node.

Example 1:



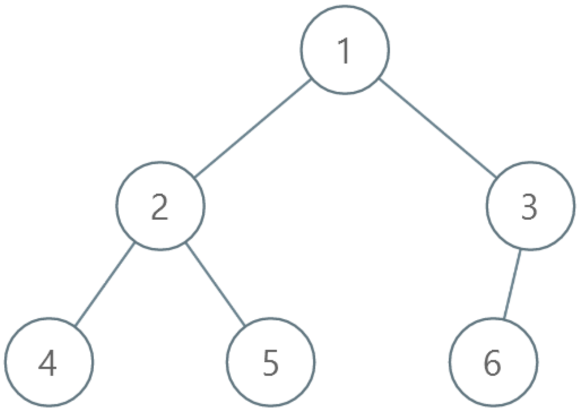
```
Input: root = [1,4,3,7,6,8,5,null,null,null,null,9,null,10]
Output: 3
Explanation:
- Swap 4 and 3. The 2nd level becomes [3,4].
- Swap 7 and 5. The 3rd level becomes [5,6,8,7].
- Swap 8 and 7. The 3rd level becomes [5,6,7,8].
We used 3 operations so return 3.
It can be proven that 3 is the minimum number of operations needed.
```

Example 2:



```
Input: root = [1,3,2,7,6,5,4]
Output: 3
Explanation:
- Swap 3 and 2. The 2nd level becomes [2,3].
- Swap 7 and 4. The 3rd level becomes [4,6,5,7].
- Swap 6 and 5. The 3rd level becomes [4,5,6,7].
We used 3 operations so return 3.
It can be proven that 3 is the minimum number of operations needed.
```

Example 3:



```
Input: root = [1,2,3,4,5,6]
Output: 0
Explanation: Each level is already sorted in increasing order so return 0.
```

Constraints:

- The number of nodes in the tree is in the range `[1, 10^5]`.
- `1 <= Node.val <= 10^5`
- All the values of the tree are **unique**.

```
def minimumOperations(self, root: Optional[TreeNode]) -> int:
        ans  = 0
        queue = [root]
        def swap(arr:list,n:int):
            nonlocal ans
            correct = sorted(arr)
            hashmap = {vall:ind for ind,vall in enumerate(arr)}
            temp = 0
            for i in range(n):
                if arr[i] != correct[i]:
                    ans+=1
                    temp = arr[i]
                    arr[i],arr[hashmap[arr[i]]]  = correct[i],arr[i]
                    hashmap[temp] = hashmap[correct[i]]
                    hashmap[correct[i]] = i
        while queue:
            n = len(queue)
            for _ in range(n):
                node = queue.pop(0)
                if node.left:
                    queue.append(node.left)
                if node.right:
                    queue.append(node.right)
            arr = [i.val for i in queue]
            swap(arr,len(arr))
                        `
        return ans
```

2476. Closest Nodes Queries in a Binary Search Tree

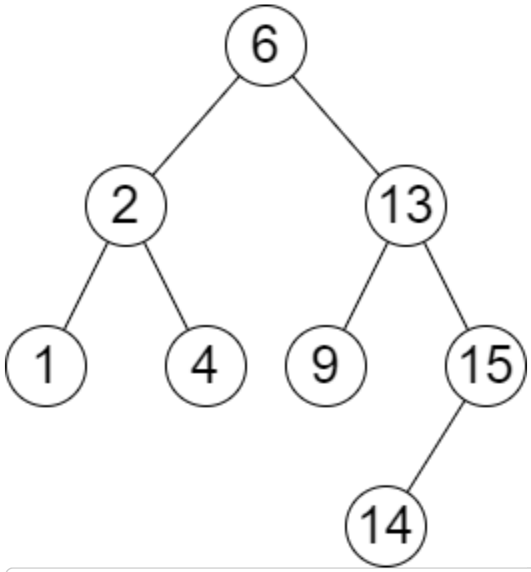
You are given the `root` of a **binary search tree** and an array `queries` of size `n` consisting of positive integers.

Find a **2D** array `answer` of size `n` where `answer[i] = [min`$_i$`, max`$_i$`]`:

- `min`$_i$ is the **largest** value in the tree that is smaller than or equal to `queries[i]`. If a such value does not exist, add `-1` instead.
- `max`$_i$ is the **smallest** value in the tree that is greater than or equal to `queries[i]`. If a such value does not exist, add `-1` instead.

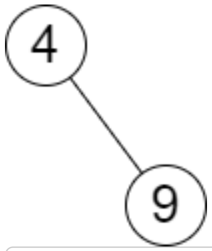
Return *the array* `answer`.

Example 1:



```
Input: root = [6,2,13,1,4,9,15,null,null,null,null,null,null,14], queries = [2,5,16]
Output: [[2,2],[4,6],[15,-1]]
Explanation: We answer the queries in the following way:
- The largest number that is smaller or equal than 2 in the tree is 2, and the smalle
- The largest number that is smaller or equal than 5 in the tree is 4, and the smalle
- The largest number that is smaller or equal than 16 in the tree is 15, and the smal
```

Example 2:



```
Input: root = [4,null,9], queries = [3]
Output: [[-1,4]]
Explanation: The largest number that is smaller or equal to 3 in the tree does not e
```

Constraints:

- The number of nodes in the tree is in the range `[2, 10^5]`.
- `1 <= Node.val <= 10^6`
- `n == queries.length`
- `1 <= n <= 10^5`
- `1 <= queries[i] <= 10^6`

[1, 4, 14, 15, 16] [1, 4, 14, 15, 16] 2 1 4 1 6 2 14 2 9 2 14 2 10 2 14 2

[[14,0],[14,0],[14,0],[14,0]]

2641. Cousins in Binary Tree II

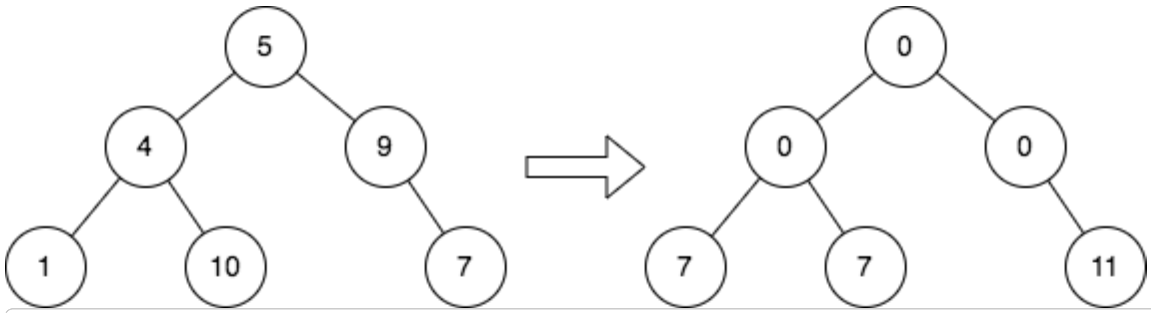
Given the `root` of a binary tree, replace the value of each node in the tree with the **sum of all its cousins' values**.

Two nodes of a binary tree are **cousins** if they have the same depth with different parents.

Return *the* `root` *of the modified tree*.

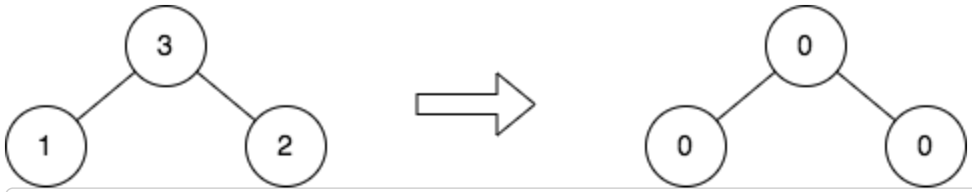
Note that the depth of a node is the number of edges in the path from the root node to it.

Example 1:



```
Input: root = [5,4,9,1,10,null,7]
Output: [0,0,0,7,7,null,11]
Explanation: The diagram above shows the initial binary tree and the binary tree afte
- Node with value 5 does not have any cousins so its sum is 0.
- Node with value 4 does not have any cousins so its sum is 0.
- Node with value 9 does not have any cousins so its sum is 0.
- Node with value 1 has a cousin with value 7 so its sum is 7.
- Node with value 10 has a cousin with value 7 so its sum is 7.
- Node with value 7 has cousins with values 1 and 10 so its sum is 11.
```

Example 2:



```
Input: root = [3,1,2]
Output: [0,0,0]
Explanation: The diagram above shows the initial binary tree and the binary tree afte
- Node with value 3 does not have any cousins so its sum is 0.
- Node with value 1 does not have any cousins so its sum is 0.
- Node with value 2 does not have any cousins so its sum is 0.
```

Constraints:

- The number of nodes in the tree is in the range `[1, 10^5]`.
- `1 <= Node.val <= 10^4`

```
class Solution:
    def replaceValueInTree(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
        queue = [[0,root]]
        hashmap = defaultdict(int)
        while queue:
            n =len(queue)
            total = 0
            for _ in range(n):
                idx,node = queue.pop(0)
                if node.left:
                    queue.append([idx*2+1,node.left])
                    hashmap[idx]+=node.left.val
                    total+=node.left.val
                if node.right:
                    queue.append([idx*2+2,node.right])
                    hashmap[idx]+=node.right.val
                    total+=node.right.val
            for i in range(len(queue)):
                if len(hashmap)<=1:
                    queue[i][1].val = 0
                    continue
                idx = (queue[i][0]-1)//2
                queue[i][1].val = total - hashmap[idx]
            hashmap.clear()
        root.val = 0
        return root
```

3067. Count Pairs of Connectable Servers in a Weighted Tree Network

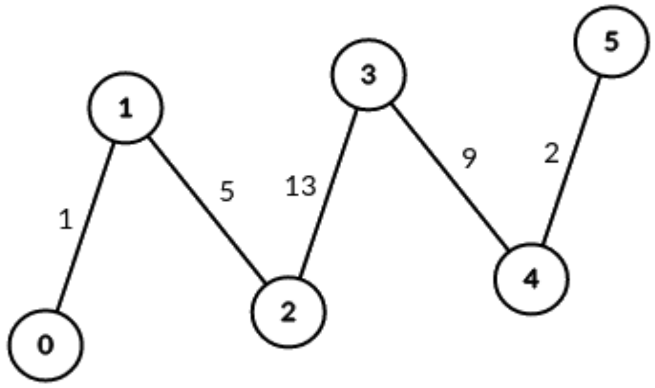
You are given an unrooted weighted tree with `n` vertices representing servers numbered from `0` to `n - 1`, an array `edges` where `edges[i] = [`a_i`, `b_i`, `$weight_i$`]` represents a bidirectional edge between vertices a_i and b_i of weight $weight_i$. You are also given an integer `signalSpeed`.

Two servers `a` and `b` are **connectable** through a server `c` if:

- `a < b`, `a != c` and `b != c`.
- The distance from `c` to `a` is divisible by `signalSpeed`.
- The distance from `c` to `b` is divisible by `signalSpeed`.
- The path from `c` to `b` and the path from `c` to `a` do not share any edges.

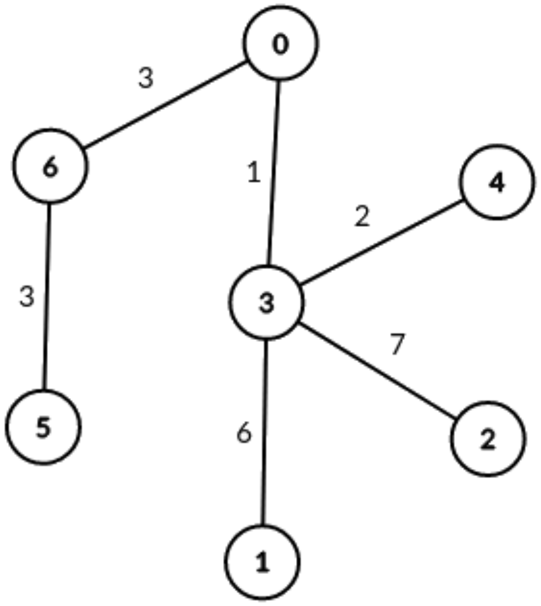
Return *an integer array* `count` *of length* `n` *where* `count[i]` *is the **number** of server pairs that are **connectable** through the server* `i`.

Example 1:



```
Input: edges = [[0,1,1],[1,2,5],[2,3,13],[3,4,9],[4,5,2]], signalSpeed = 1
Output: [0,4,6,6,4,0]
Explanation: Since signalSpeed is 1, count[c] is equal to the number of pairs of path
In the case of the given path graph, count[c] is equal to the number of servers to th
```

Example 2:



```
Input: edges = [[0,6,3],[6,5,3],[0,3,1],[3,2,7],[3,1,6],[3,4,2]], signalSpeed = 3
Output: [2,0,0,0,0,0,2]
Explanation: Through server 0, there are 2 pairs of connectable servers: (4, 5) and (
Through server 6, there are 2 pairs of connectable servers: (4, 5) and (0, 5).
It can be shown that no two servers are connectable through servers other than 0 and
```

Constraints:

- `2 <= n <= 1000`
- `edges.length == n - 1`
- `edges[i].length == 3`
- `0 <=` a_i, b_i `< n`
- `edges[i] = [`a_i, b_i, $weight_i$`]`
- `1 <=` $weight_i$ `<=` 10^6
- `1 <= signalSpeed <=` 10^6
- The input is generated such that `edges` represents a valid tree.

```
def dfs(self,node,visited,distance,signalspeed,graph):
    num = 0
    if distance%signalspeed ==0:num += 1
    visited.add(node)
    for child,weight in graph[node]:
        if child not in visited:
            child_nums = self.dfs(child,visited,distance+weight,signalspeed,graph)
            num += child_nums
    visited.remove(node)
    return num

def countPairsOfConnectableServers(self, edges: List[List[int]], signalSpeed: int)
-> List[int]:
    n = len(edges)+1
    graph = defaultdict(list)
    for a,b,w in edges:
        graph[a].append([b,w])
        graph[b].append([a,w])
    ans  = [0]*n
    for node in range(n):
        num = []
        for child,weight in graph[node]:
            child_nums = self.dfs(child,set([node]),weight,signalSpeed,graph)
            num.append(child_nums)
        s = sum(num)
        for i in range(len(num)):
            ans[node] += (s-num[i]) * num[i]
        ans[node] = ans[node]//2
    return ans
```

3319. K-th Largest Perfect Subtree Size in Binary Tree

You are given the `root` of a **binary tree** and an integer `k`.

Return an integer denoting the size of the k^{th} **largest perfect binary** subtree, or `-1` if it doesn't exist.

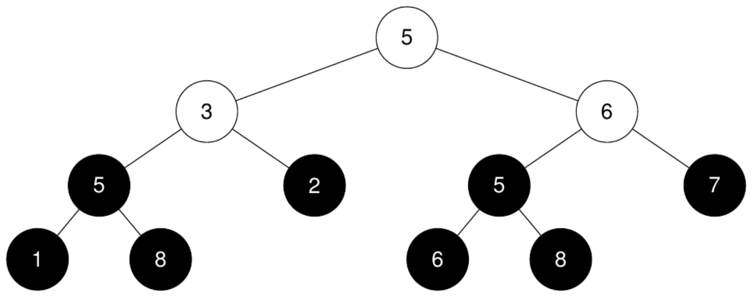
A **perfect binary tree** is a tree where all leaves are on the same level, and every parent has two children.

Example 1:

Input: root = [5,3,6,5,2,5,7,1,8,null,null,6,8], k = 2

Output: 3

Explanation:



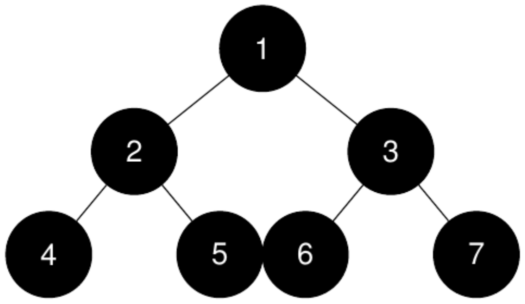
The roots of the perfect binary subtrees are highlighted in black. Their sizes, in non-increasing order are `[3, 3, 1, 1, 1, 1, 1, 1]`.
The 2nd largest size is 3.

Example 2:

Input: root = [1,2,3,4,5,6,7], k = 1

Output: 7

Explanation:



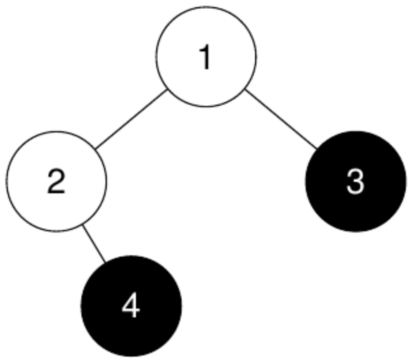
The sizes of the perfect binary subtrees in non-increasing order are `[7, 3, 3, 1, 1, 1, 1]`. The size of the largest perfect binary subtree is 7.

Example 3:

Input: root = [1,2,3,null,4], k = 3

Output: -1

Explanation:



The sizes of the perfect binary subtrees in non-increasing order are `[1, 1]`. There are fewer than 3 perfect binary subtrees.

Constraints:

- The number of nodes in the tree is in the range `[1, 2000]`.
- `1 <= Node.val <= 2000`
- `1 <= k <= 1024`

```
def kthLargestPerfectSubtree(self, root: Optional[TreeNode], k: int) -> int:
        if not root:
                return -1
        ans = []
        def dfs(root):
                nonlocal ans
                l = 0
                r = 0
                if root.left:
                        l = dfs(root.left)
                if root.right:
                        r = dfs(root.right)
                if not root.left and not root.right:
                        ans.append(1)
                        return 1
                elif l == r and l>0:
                        ans.append(l+r+1)
                        return l+r+1
                return 0
        dfs(root)
        ans.sort(reverse = True)
        return ans[k-1] if len(ans)>k-1 else -1
```

3331. Find Subtree Sizes After Changes

You are given a tree rooted at node 0 that consists of `n` nodes numbered from `0` to `n - 1`. The tree is represented by an array `parent` of size `n`, where `parent[i]` is the parent of node `i`. Since node 0 is the root, `parent[0] == -1`.

You are also given a string `s` of length `n`, where `s[i]` is the character assigned to node `i`.

We make the following changes on the tree **one** time **simultaneously** for all nodes `x` from `1` to `n - 1`:

- Find the **closest** node `y` to node `x` such that `y` is an ancestor of `x`, and `s[x] == s[y]`.
- If node `y` does not exist, do nothing.
- Otherwise, **remove** the edge between `x` and its current parent and make node `y` the new parent of `x` by adding an edge between them.

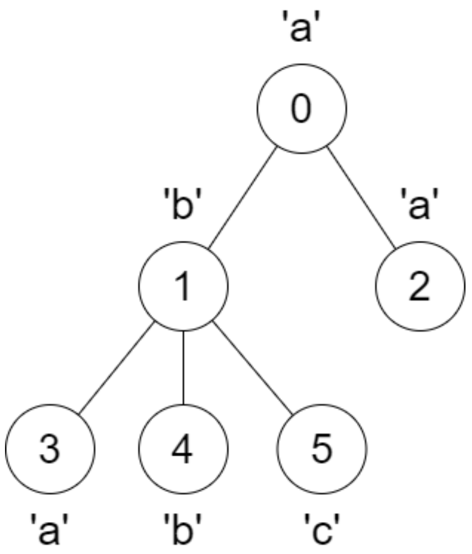
Return an array `answer` of size `n` where `answer[i]` is the **size** of the subtree rooted at node `i` in the **final** tree.

Example 1:

Input: parent = [-1,0,0,1,1,1], s = "abaabc"

Output: [6,3,1,1,1,1]

Explanation:



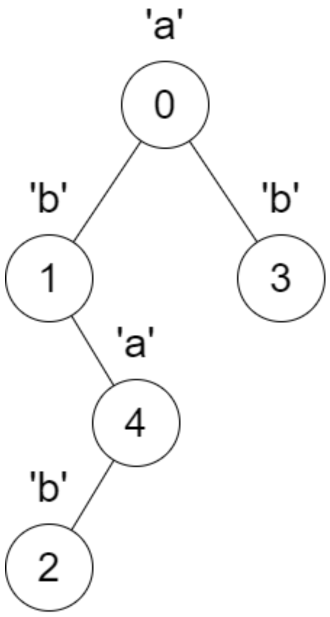
The parent of node 3 will change from node 1 to node 0.

Example 2:

Input: parent = [-1,0,4,0,1], s = "abbba"

Output: [5,2,1,1,1]

Explanation:



The following changes will happen at the same time:

- The parent of node 4 will change from node 1 to node 0.
- The parent of node 2 will change from node 4 to node 1.

Constraints:

- `n == parent.length == s.length`
- `1 <= n <=` 10^5
- `0 <= parent[i] <= n - 1` for all `i >= 1`.
- `parent[0] == -1`
- `parent` represents a valid tree.
- `s` consists only of lowercase English letters.

```
class Solution:
    def findSubtreeSizes(self, p: List[int], s: str) -> List[int]:
        n, clds = len(s), defaultdict(set)
        for i in range(n):
            clds[p[i]].add(i)
        res = [0] * n
        def dfs(i, anc={}):
            prv, anc[s[i]] = anc.get(s[i]), i
            res[i] = 1
            for c in clds[i]:
                dfs(c, anc)
                res[p[c]] += res[c]
            anc[s[i]] = prv
            if prv is not None:
                p[i] = prv

        dfs(0)
        return res
```

3372. Maximize the Number of Target Nodes After Connecting Trees I

There exist two **undirected** trees with `n` and `m` nodes, with **distinct** labels in ranges `[0, n - 1]` and `[0, m - 1]`, respectively.

You are given two 2D integer arrays `edges1` and `edges2` of lengths `n - 1` and `m - 1`, respectively, where `edges1[i] = [`a_i`, `b_i`]` indicates that there is an edge between nodes a_i and b_i in the first tree and `edges2[i] = [`u_i`, `v_i`]` indicates that there is an edge between nodes u_i and v_i in the second tree. You are also given an integer `k`.

Node `u` is **target** to node `v` if the number of edges on the path from `u` to `v` is less than or equal to `k`. **Note** that a node is *always* **target** to itself.

Return an array of `n` integers `answer`, where `answer[i]` is the **maximum** possible number of nodes **target** to node `i` of the first tree if you have to connect one node from the first tree to another node in the second tree.

Note that queries are independent from each other. That is, for every query you will remove the added edge before proceeding to the next query.

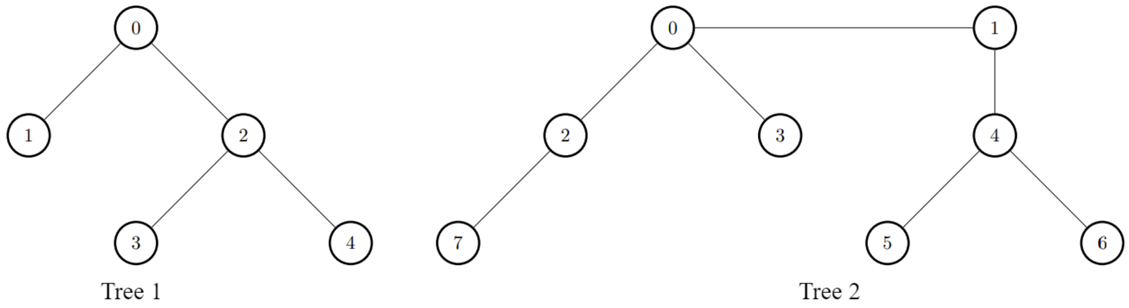
Example 1:

Input: edges1 = [[0,1],[0,2],[2,3],[2,4]], edges2 = [[0,1],[0,2],[0,3],[2,7],[1,4],[4,5],[4,6]], k = 2

Output: [9,7,9,8,8]

Explanation:

- For `i = 0`, connect node 0 from the first tree to node 0 from the second tree.
- For `i = 1`, connect node 1 from the first tree to node 0 from the second tree.
- For `i = 2`, connect node 2 from the first tree to node 4 from the second tree.
- For `i = 3`, connect node 3 from the first tree to node 4 from the second tree.
- For `i = 4`, connect node 4 from the first tree to node 4 from the second tree.



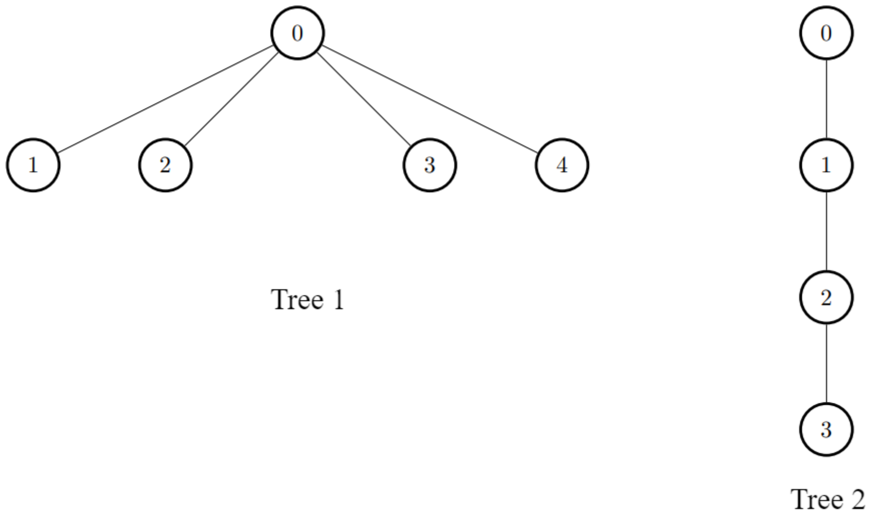
Example 2:

Input: edges1 = [[0,1],[0,2],[0,3],[0,4]], edges2 = [[0,1],[1,2],[2,3]], k = 1

Output: [6,3,3,3,3]

Explanation:

For every `i`, connect node `i` of the first tree with any node of the second tree.



Constraints:

- `2 <= n, m <= 1000`

- `edges1.length == n - 1`
- `edges2.length == m - 1`
- `edges1[i].length == edges2[i].length == 2`
- `edges1[i] = [`a_i`, `b_i`]`
- `0 <= `a_i`, `b_i` < n`
- `edges2[i] = [`u_i`, `v_i`]`
- `0 <= `u_i`, `v_i` < m`
- The input is generated such that `edges1` and `edges2` represent valid trees.
- `0 <= k <= 1000`

```
class Solution:
    def makegraph(self,mapp,edges):
        for i,j in edges:
            mapp[i].append(j)
            mapp[j].append(i)
        return mapp
    def do_bfs_traversal(self,mapp,k):
        reachCounts  = [0]*len(mapp)
        for startnode in range(len(mapp)):
            queue = [startnode]
            visited = set()
            visited.add(startnode)
            level = 0
            while queue and level < k :
                level+=1
                n = len(queue)
                for _ in range(n):
                    node = queue.pop(0)
                    for j in mapp[node]:
                        if j not in visited:
                            visited.add(j)
                            queue.append(j)
            reachCounts[startnode] += len(visited)
        return reachCounts

    def maxTargetNodes(self, edges1: List[List[int]], edges2: List[List[int]], k: i
nt) -> List[int]:
        if k == 0:
            return [1] * (len(edges1) + 1)
        tree1 = self.makegraph(defaultdict(list),edges1)
        tree2 = self.makegraph(defaultdict(list),edges2)
        reachableCount1 = self.do_bfs_traversal(tree1,k)
        reachableCount2 = self.do_bfs_traversal(tree2,k-1)
        maxReachableInTree2 = max(reachableCount2)
        for i in range(len(reachableCount1)):
            reachableCount1[i] += maxReachableInTree2
        return reachableCount1

    ````
```

## 3373. Maximize the Number of Target Nodes After Connecting Trees II
```

There exist two **undirected** trees with `n` and `m` nodes, labeled from `[0, n - 1]` and `[0, m - 1]`, respectively.

You are given two 2D integer arrays `edges1` and `edges2` of lengths `n - 1` and `m - 1`, respectively, where `edges1[i] = [a_i, b_i]` indicates that there is an edge between nodes `a_i` and `b_i` in the first tree and `edges2[i] = [u_i, v_i]` indicates that there is an edge between nodes `u_i` and `v_i` in the second tree.

Node `u` is **target** to node `v` if the number of edges on the path from `u` to `v` is even. **Note** that a node is *always* **target** to itself.

Return an array of `n` integers `answer`, where `answer[i]` is the **maximum** possible number of nodes that are **target** to node `i` of the first tree if you had to connect one node from the first tree to another node in the second tree.

Note that queries are independent from each other. That is, for every query you will remove the added edge before proceeding to the next query.

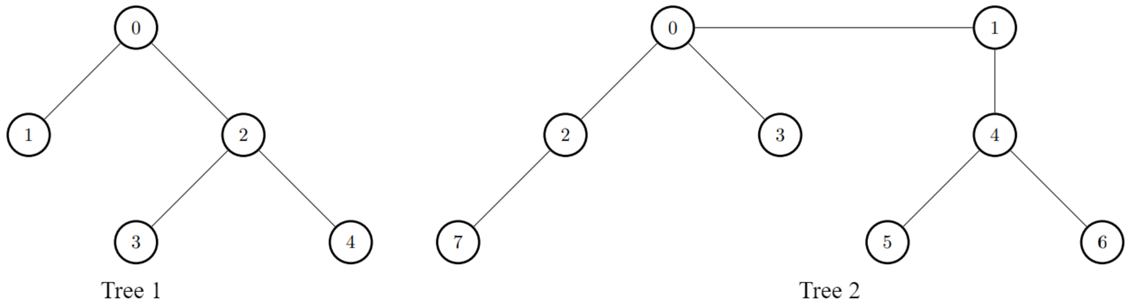
Example 1:

Input: edges1 = [[0,1],[0,2],[2,3],[2,4]], edges2 = [[0,1],[0,2],[0,3],[2,7],[1,4],[4,5],[4,6]]

Output: [8,7,7,8,8]

Explanation:

- For `i = 0`, connect node 0 from the first tree to node 0 from the second tree.
- For `i = 1`, connect node 1 from the first tree to node 4 from the second tree.
- For `i = 2`, connect node 2 from the first tree to node 7 from the second tree.
- For `i = 3`, connect node 3 from the first tree to node 0 from the second tree.
- For `i = 4`, connect node 4 from the first tree to node 4 from the second tree.



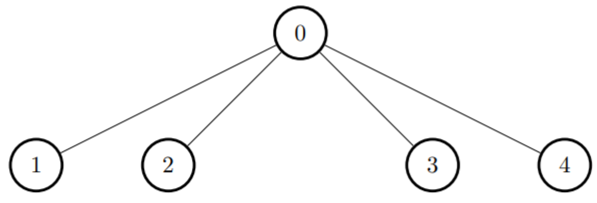
Example 2:

Input: edges1 = [[0,1],[0,2],[0,3],[0,4]], edges2 = [[0,1],[1,2],[2,3]]

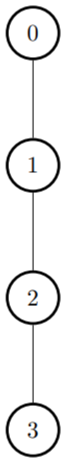
Output: [3,6,6,6,6]

Explanation:

For every `i`, connect node `i` of the first tree with any node of the second tree.



Tree 1



Tree 2

Constraints:

- `2 <= n, m <=` 10^5
- `edges1.length == n - 1`
- `edges2.length == m - 1`
- `edges1[i].length == edges2[i].length == 2`
- `edges1[i] = [`a_i`,` b_i`]`
- `0 <=` a_i`,` b_i `< n`
- `edges2[i] = [`u_i`,` v_i`]`
- `0 <=` u_i`,` v_i `< m`
- The input is generated such that `edges1` and `edges2` represent valid trees.

---

graph colouring very important

```
class Solution:
    def makegraph(self,mapp,arr):
        for i,j in arr:mapp[i].append(j);mapp[j].append(i)
        return mapp
    def color(self,tree):
        queue = [0]
        visited = {0:"b"}
        level = True
        black,white = 1,0
        while queue:
            level = not level
            n = len(queue)
            for _ in range(n):
                node = queue.pop(0)
                for i in tree[node]:
                    if i not in visited:
                        queue.append(i)
                        if level :
                            black+=1
                            visited[i] = "b"
                        else:
                            white+=1
                            visited[i] = "w"
        return black,white,visited

    def maxTargetNodes(self, edges1: List[List[int]], edges2: List[List[int]]) -> L
ist[int]:
        tree1 = self.makegraph(defaultdict(list),edges1)
        tree2 = self.makegraph(defaultdict(list),edges2)
        black1,white1,vis1 = self.color(tree1)
        black2,white2,vis2 = self.color(tree2)
        maxcol = max(black2,white2)
        ans = []
        for i in range(len(tree1)):
            col = maxcol
            if vis1[i] == "b":col+= black1
            else:col+= white1
            ans.append(col)
        return ans
```