1519. Number of Nodes in the Sub-Tree With the Same Label **Breadth-First Search** Medium Tree **Depth-First Search** Hash Table Counting **Leetcode Link**

Problem Description In this problem, we are given a tree consisting of n nodes. Each node has a unique number from 0 to n-1 and a label represented by a

of nodes indicating there is an edge between those nodes. The goal is to count how many nodes in each node's subtree (including the node itself) have the same label as that particular node and then return the counts in an array. Intuition

lowercase character from the given string labels. The tree has n-1 edges provided in the edges array, where each element is a pair

To arrive at the solution for this problem, we should consider a depth-first search (DFS) strategy since we want to deal with subtrees and descendants in a tree structure. The DFS will start from the root node (node 0), explore as far as possible along each branch before backtracking, and perform the necessary counting operations.

own subtree.

1. We use DFS to traverse the entire tree from the root node, exploring all its subtrees. 2. We maintain a counter, cnt, using a Counter object to track the occurrence of each label as we move down the tree. 3. While performing DFS, for each node, we decrement the count for its label before the DFS goes deeper into its children. This is done to keep track of how many times the label occurs in the current node's subtree. 4. After visiting all of the children nodes, we increment the count back, thus accounting for the occurrence of the node's label in its

5. The counter differences before and after exploring the children will give us the number of children with the same label.

Here's the intuition behind the provided solution:

6. Create a list g to represent the adjacency list for the graph. For each edge, we add nodes to each other's list to create this

count the nodes, and backtrack to the parent nodes while updating the counts.

• We increment cnt[labels[i]] as we've encountered node i's label.

node i for each node. This array is returned as the result.

- undirected graph representation. 7. Create an array ans of size n to store the counts for each node. 8. Call the dfs function with the root node and an invalid parent index -1 to kick-start the DFS.
- **Solution Approach**
- 9. The updated ans array will be returned, which contains the count of nodes in the subtree of every node with the same label as that node.
- The solution uses recursion and graph traversal techniques to address the problem effectively. Here's a breakdown of the solution implementation process with the algorithms, data structures, and patterns employed: 1. Depth-First Search (DFS): A classic DFS recursion is the crux of traversing the tree. The function dfs takes two parameters, i

representing the current node, and fa representing the parent of the current node. The recursion helps us reach the leaves,

2. Counter Object: We use a Counter object from the collections module, named cnt. It keeps track of the count of labels

encountered during the DFS traversal. It plays a pivotal role in calculating the number of nodes bearing the same label within a subtree.

has no parent.

cnt = Counter()

def dfs(i, fa):

14

15

16

19 dfs(0, -1)

as that node.

0(a)

1(a) 2(b)

1 g[0] = [1, 2]

q[2] = [0]

g[3] = [1]

5 g[4] = [1]

g[1] = [0, 3, 4]

5 3(b) 4(a)

Recursive DFS Function

for j in g[i]:

cnt[labels[i]] += 1

if j != fa:

18 # Start DFS from root node 0

ans[i] -= cnt[labels[i]]

dfs(j, i)

First, the tree's structure can be visualized as shown below:

1. Create the adjacency list g for graph representation:

According to the labels string, the labels of nodes in the order from 0 to 4 are a, a, b, b, a.

• ans [0] is decremented by cnt['a'] (which is 0 at this moment), so ans [0] remains 0.

■ ans [1] is decremented by cnt ['a'] (which is 1), so ans [1] = -1.

DFS is called on its children (none), so no further action.

ans [3] is incremented by cnt ['b'] (1), becomes 1.

ans [4] is decremented by cnt['a'] (2), becomes -2.

ans [4] is incremented by cnt['a'] (3), becomes 1.

ans [2] is decremented by cnt ['b'] (1), becomes -1.

After its children are done, ans [1] is incremented by cnt ['a'] (3), becomes 2.

After visiting all children of node 0, ans [0] is incremented by cnt['a'] (3), becomes 3.

def countSubTrees(self, n: int, edges: List[List[int]], labels: str) -> List[int]:

Recursive function to traverse the graph and update counts

Decrement the count of the label for this node

label_counts[labels[node]] -= 1

for adjacent in graph[node]:

if adjacent != parent:

Traverse adjacent nodes (children)

Create a graph from the edges (adjacency list)

List to store the result for each node

// Graph represented as a list of adjacency lists.

// The labels for each node are stored in a string.

// Count array to store the frequency of each character.

// Array to store the final answer for each node.

int from = edge[0], to = edge[1];

// Set the labels and prepare the answer array.

// Start DFS traversal from the first node.

// Private helper method to perform DFS traversal.

int labelIndex = labels.charAt(node) - 'a';

// Get the index of the label character for the current node.

// Increment the count for this character as this node is visited.

// all occurrences in the subtree rooted at this node.

// Visit all the connected nodes that are not the parent.

// Decrement count of the current label at this node before DFS, as this count will include

private void dfs(int node, int parent) {

answer[node] -= count[labelIndex];

for (int neighbor : graph[node]) {

dfs(neighbor, node);

if (neighbor != parent) {

Counter to keep track of the label frequencies during DFS

Call DFS starting from the root node (0) with no parent (-1)

After DFS, results list contains the counts for nodes' labels as required

// Public method that initializes the class variables and starts the DFS traversal.

ans[i] += cnt[labels[i]]

ans = [0] * n

4. **Recursive Logic**: In the dfs function:

3. Adjacency List Representation: A dictionary named g is used to store the adjacency list. Here, we map each node to a list containing its neighboring nodes. This data structure allows us to represent the undirected graph for the tree efficiently.

 We loop over each child j in the adjacency list for node i and perform a DFS call if j is not the parent (to avoid revisiting the parent). • After visiting all children, we add back the cnt[labels[i]] count to ans[i], which now contains the correct count for the subtree at i.

5. Result: The ans array, which initially is set to zeros of length n, is updated with the count of nodes that have the same label as

6. Initialization and Start Point: Before starting the DFS traversal, we need to initialize the adjacency list g and the result array ans.

We populate g based on the edges provided. The dfs function is called with the root node (0) and -1 as the parent since the root

• First, we subtract cnt[labels[i]] from ans[i] to record the initial count of the label for node i.

Initialization = defaultdict(list) for a, b in edges: g[a].append(b) g[b].append(a)

Here is the code block relevant to the above explanation, divided into initialization and recursive DFS parts for clarity:

By maintaining a counter that is updated at each node of the DFS, we ensure that when we calculate the count for node i, we have a complete picture of the number of times its label appears within its entire subtree. **Example Walkthrough**

Consider a small example where we have a tree with n = 5 nodes. The edges are given by edges = [[0, 1], [0, 2], [1, 3], [1,

4]], and the labels string is "ababa". Using these, we need to count the number of nodes in each node's subtree with the same label

2. Initialize a Counter object, cnt, and an array ans of size n: 1 cnt = Counter()

For node 1:

During the first call dfs(0, -1):

• cnt['a'] is incremented to 1.

Recursive call dfs(1, 0):

dfs(4, 1) is called:

We start with our solution approach:

2 ans = [0, 0, 0, 0, 0]3. We start DFS from the root node 0 with -1 as the parent:

cnt['a'] is incremented to 2. dfs(3, 1) is called: ans [3] is decremented by cnt['b'] (0), remains 0.

cnt['b'] is incremented to 1.

• We then call DFS on its children (nodes 1 and 2):

- cnt['a'] is incremented to 3. DFS is called on its children (none), so no further action.
- cnt['b'] is incremented to 2. DFS is called on its children (none), so no further action. ans [2] is incremented by cnt ['b'] (2), becomes 1.

Recursive call dfs(2, 0):

For node 2:

their subtrees:

1 ans = [3, 2, 1, 1, 1]

Python Solution

class Solution:

10

11

12

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

8

9

10

11

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

36

37

38

39

40

41

43

10

12

13

14

15

16

24

25

26

27

28

29

31

32

33

34

35

37

38

40

39 }

42 };

};

// Start DFS from node 0 with no parent

// Function to count the number of occurrences of labels in the subtrees of each node

// Determine the index (0-25) corresponding to the label's character

// Subtract the current count before the DFS of the children to later find the delta

if (nextNode !== parent) { // Skip the parent node to prevent going backwards

function countSubTrees(n: number, edges: number[][], labels: string): number[] {

const depthFirstSearch = (currentNode: number, parent: number) => {

const labelIndex = labels.charCodeAt(currentNode) - 97;

subtreeCounts[currentNode] -= labelCounts[labelIndex];

for (const nextNode of adjacencyList[currentNode]) {

depthFirstSearch(nextNode, currentNode);

// After visiting children, add the updated count to the result

// Array to hold the counts of each label globally across the DFS traversal

// Adjacency list to represent the graph, with an array for each node

const adjacencyList: number[][] = Array.from({ length: n }, () => []);

// Start DFS traversal from the node with label '0', with no parent (-1)

// Return the array containing counts of each label in the subtree of each node

// Traverse all the connected nodes (children)

const labelCounts: number[] = new Array(26).fill(0);

// Construct the adjacency list from the edges

for (const [nodeA, nodeB] of edges) {

depthFirstSearch(0, -1);

return subtreeCounts;

Space Complexity

adjacencyList[nodeA].push(nodeB);

adjacencyList[nodeB].push(nodeA);

depthFirstSearch(0, -1);

// DFS function to traverse the nodes

// Increase the character count

labelCounts[labelIndex]++;

return answer;

Typescript Solution

from typing import List

1 from collections import defaultdict, Counter

def dfs(node, parent):

graph = defaultdict(list)

graph[a].append(b)

graph[b].append(a)

label_counts = Counter()

private List<Integer>[] graph;

for (int[] edge : edges) {

graph[from].add(to);

graph[to].add(from);

this.labels = labels;

answer = new int[n];

count = new int[26];

count[labelIndex]++;

dfs(0, -1);

return answer;

private String labels;

private int[] answer;

private int[] count;

for a, b in edges:

results = [0] * n

dfs(0, -1)

return results

13 dfs(adjacent, node) # Increment the count of the label for this node to include itself 14 label_counts[labels[node]] += 1 15 # Update the result for this node 16 # (Total count of the label minus the count before visiting the subtree) 17 results[node] = label_counts[labels[node]]

So, after the DFS completes, we have the ans array updated with the count of the nodes that have the same label as each node in

public int[] countSubTrees(int n, int[][] edges, String labels) { 12 13 // Initialize graph with empty lists for each node. 14 graph = new List[n]; 15 Arrays.setAll(graph, x -> new ArrayList<>()); 16 17 // Populating the adjacency list for the undirected graph.

Java Solution

class Solution {

```
49
 50
 51
 52
             // After visiting all children, increment the answer for this node.
 53
             // Now the count includes all occurrences in subtree plus the current node.
             answer[node] += count[labelIndex];
 54
 55
 56
 57
C++ Solution
 1 class Solution {
   public:
       // Method to count the number of subtrees with the same label for each node
       vector<int> countSubTrees(int n, vector<vector<int>>& edges, string labels) {
           // Create an adjacency list representation of the graph
           vector<vector<int>> graph(n);
           for (auto& edge : edges) {
               int from = edge[0], to = edge[1];
               graph[from].push_back(to);
9
               graph[to].push_back(from);
10
11
12
           // Vector to store the answer
           vector<int> answer(n);
14
15
16
           // Array to count occurrences of each character 'a' to 'z'
17
           int charCount[26] = {0};
18
19
           // Define the DFS function
           function<void(int, int)> depthFirstSearch = [&](int node, int parent) {
20
               int charIndex = labels[node] - 'a'; // Convert char to index (0 - 25)
21
22
23
               // Subtract current count from the answer for this node, as it will be 'recounted' during backtracking
24
               answer[node] -= charCount[charIndex];
25
               charCount[charIndex]++; // Increment the count of the current character
26
27
               // Recurse for all adjacent nodes
28
               for (int& adjacent : graph[node]) {
                   if (adjacent != parent) { // Avoid revisiting the parent node
29
30
                       depthFirstSearch(adjacent, node);
31
32
33
               // Add the total count discovered during the recursive calls to the answer for this node
34
35
               answer[node] += charCount[charIndex];
```

subtreeCounts[currentNode] += labelCounts[labelIndex]; 18 **}**; 19 20 21 // Array to hold the counts of each label's occurrences in the subtree rooted at each node 22 const subtreeCounts: number[] = new Array(n).fill(0);

```
Time and Space Complexity
Time Complexity
The time complexity of the code is O(n) where n is the number of nodes in the tree. Each node is visited exactly once due to the
depth-first search (DFS). Inside each DFS call, operations are constant time except for the iteration over the children which, across
all calls, account for n - 1 edges. Since every edge connects two nodes and each edge is visited exactly twice (once from each
node it connects), the time complexity due to edges is 0(n - 1) which simplifies to 0(n). Thus, the overall time complexity is
governed by the number of node visits and edge traversals combined, which is linear in the number of nodes.
```

The space complexity is O(n). The dictionary g storing the graph can contain up to O(n-1) entries because it is an undirected graph and each edge is stored twice. The cnt Counter and ans array each require O(n) space. The space required for the dfs call stack can also go up to O(n) in the case of a skewed tree where the recursive calls are not returned until the end of the longest branch. Therefore, the total space complexity combines the space for graph representation, counters, answer storage, and the recursion stack, which all scale linearly with the number of nodes in the tree.