2049. Count Nodes With the Highest Score

Problem Description

This problem presents a binary tree rooted at node `0` and consisting of `n` nodes, each uniquely labeled from `0` to `n - 1`. We are given an integer array `parents` where each element represents the parent of a corresponding node, such that `parents[i]` is the parent of node `i`. The root node's parent is indicated by `parents[0]` being `-1`.

The score of a node is determined by the product of the number of nodes in each of its connected edges from the tree. This operation results in a number of non-empty subtrees. The size of each subtree is simply the number of nodes it contains. The score for the removed node is calculated as the product of the sizes of all resultant subtrees. The goal is to find out how many nodes in the tree have the highest score.

Intuition

To solve this problem, we need a way to compute the score of each node efficiently. Instead of actually removing nodes, which would be inefficient, we can simulate the process with a depth-first search (DFS) algorithm. Here's the logic behind the solution:

- A tree structure often calls for a recursive strategy. DFS is a good fit here as we need to explore subtrees to calculate scores.
- The solution tracks the size of each subtree as the DFS traversal visits the nodes. The size of the subtree rooted at a node is 1 (for the node itself) plus the sizes of all subtrees of its children.
- If a node is removed, the score for that node is the product of the sizes of the subtrees left behind. However, there is one additional "subtree" to consider — the one formed by the rest of the tree outside of this node's subtree. Its size is the total number of nodes in the tree minus the size of the subtree rooted at the given node.
- For the root node, its score is simply the product of the sizes of its child subtrees because there is no "parent" subtree to consider.
- In the recursive DFS function, we track the maximum score found and count how many times nodes with this score occur.
- For each node visited, we calculate a temporary score. If it's greater than the current maximum score, we update the maximum score and reset our count of nodes with the maximum score to 1. If the score matches the current maximum, we increment our count.
- After the DFS completes, the counter will have the number of nodes with the highest score.

Putting it all together, starting the DFS from the root will give us the needed information to return the final count of nodes with the highest score.

Solution Approach

The solution employs a Depth-First Search (DFS) algorithm to compute subtree sizes and uses a simple list (Python's `list` data structure) to model the graph (tree) structure.

1. **Transformation into a graph**: The given `parents` array is used to create an adjacency list that represents the tree. For each node (excluding the root), we append it to a list at the index of its parent. This means `g[i]` will contain a list of all children of the `i`th node.
2. **DFS algorithm**: We define a recursive function `dfs` that takes a single argument `cur`, which represents the current node being visited. The function initializes a local variable `size` to `1` for the current node and `score` to `1` (the initial product of subtree sizes).
3. **Calculating subtree sizes and scores**: For every child `c` of the current node `cur`, we recursively call `dfs(c)`, which returns the size of the subtree rooted at `c`. This value is added to the `size` of the current subtree and also multiplied into the `score` for the current node.
4. **Handling of the non-subtree part**: If the current node is not the root, we need to consider the rest of the tree outside of the current node's subtree. We multiply the current `score` by `n - size` to incorporate this.
5. **Tracking the maximum score and count**: We use global variables `max_score` and `ans` to keep track of the maximum score found so far, and the number of nodes with that score, respectively. If the score of the current node is higher than the `max_score`, we update `max_score` and set `ans` to `1`. If the score is equal to `max_score`, we increment `ans`.
6. **Starting the DFS and returning the result**: The DFS is started by calling `dfs(0)` because the root is labeled `0`. Once DFS is completed, `ans` will hold the count of nodes with the highest score, and this is what is returned.

The elegance of the solution comes from its efficient traversal of the tree using DFS to calculate the scores for all nodes without any modifications to the tree's structure, and clever use of global variables to keep track of the maximum score observed, as well as the count of nodes that achieve this score.

Example Walkthrough

To illustrate the solution approach, let's consider a small tree with `n = 5` nodes, and the following `parents` array representing the tree:

```
Node index:  0  1  2  3  4
Parents:    [-1, 0, 0, 1, 1]
```

This represents a tree where:

- Node `0` is the root (as `parents[0] = -1`).
- Nodes `1` and `2` are children of node `0`.
- Nodes `3` and `4` are children of node `1`.

First, we build the adjacency list based on the `parents` array to represent the tree graph:

```
g[0] => [1, 2]
g[1] => [3, 4]
g[2] => []
g[3] => []
g[4] => []
```

Now we run the DFS algorithm starting from the root node (`0`). We also initialize our global variables `max_score` and `ans` to track the maximum score and number of nodes with that maximum score.

1. Start DFS at root node 0. Initialize `size = 1`, `score = 1`.
2. For every child of node `0`, that is nodes `1` and `2`, we call `dfs(child)`.
3. Calling `dfs(1)`:
   - Start with `size = 1`, `score = 1`.
   - Visit children of node `1`, which are nodes `3` and `4`.
   - Call `dfs(3)`, which returns `1` because it has no children, adding this to `size` of node `1` and setting `score = score * size_subtree(3) = 1`.
   - Call `dfs(4)`, similar to `dfs(3)`, return `1`, adding to `size` of node `1`, and `score = score * size_subtree(4) = 1`.
   - Final `size` of subtree at node `1` is `3`, and the `score` for node `1` when removing it would be `score = (n - size) * 1 * (5 - 3) = 2`. Update `max_score` to `2` and `ans` to `1`.
4. Back to node `0`, now calling `dfs(2)`:
   - Node `2` has no children, so it returns a size of `1`.
   - Add `1` to size of node `0` which is now `3` (the total size of the tree).
5. With DFS complete for node `0`, node `1` score would be the product of sizes of its children, which are nodes `3` and `2`. So `score = size_subtree(1) * size_subtree(2) = 3 * 1 = 3`. The `max_score` is less than `3`, so we update `max_score` to `3` and reset `ans` to `1`.
6. DFS has now visited all nodes. Since no other nodes will have a score higher than `3` (the score when the root is removed), we conclude that the highest score is `3`, and there is only `1` node (the root, node `0`) with this score.

Based on the solution approach, we return `ans`, which is `1`, indicating there is one node (the root) with the highest score in the tree.

Python Solution

```python
from typing import List

class Solution:
    def countHighestScoreNodes(self, parents: List[int]) -> int:
        # Initialize the number of nodes, maximum score, and answer counter.
        num_nodes = len(parents)
        max_score = 0
        answer = 0

        # Graph representation of the tree, with each node pointing to its children.
        graph = [[] for _ in range(num_nodes)]

        # Build the adjacency list for each parent.
        for i in range(1, num_nodes):
            graph[parents[i]].append(i)

        # Depth-first search to compute subtree sizes and scores.
        def dfs(current_node) -> int:
            nonlocal max_score, answer
            subtree_size = 1
            node_score = 1

            # Visit each child and calculate the score contribution of its subtree.
            for child in graph[current_node]:
                child_subtree_size = dfs(child)
                node_score *= child_subtree_size
                subtree_size += child_subtree_size

            # If not the root, multiply by the size of the "complement" subtree.
            if current_node != 0:
                node_score *= num_nodes - subtree_size

            # Update answer and max_score in case of new max or equal score found.
            if node_score > max_score:
                max_score = node_score
                answer = 1
            elif node_score == max_score:
                answer += 1

            return subtree_size

        # Start DFS from the root of the tree (node 0).
        dfs(0)

        # Return the number of nodes that have the maximum score.
        return answer
```

Java Solution

```java
class Solution {
    private int nodeCount; // Total number of nodes in the tree
    private long maxNodeScore; // The highest score found in the tree
    private long countMaxScoreNodes; // The count of nodes having the maximum score
    private List<List<Integer>> childrenList; // Adjacency list representation of the tree

    public int countHighestScoreNodes(int[] parents) {
        nodeCount = parents.length;
        maxNodeScore = 0;
        countMaxScoreNodes = 0;
        childrenList = new ArrayList<>();

        // Initialize lists to hold children nodes for each node
        for (int i = 0; i < nodeCount; ++i) {
            childrenList.add(new ArrayList<>());
        }

        // Build the adjacency list (tree graph) from the parent array
        for (int i = 1; i < nodeCount; ++i) {
            childrenList.get(parents[i]).add(i);
        }

        // Start depth-first search from the root node (0)
        dfs(0);

        // Return the count of nodes that have the highest score
        return (int) countMaxScoreNodes;
    }

    // A method to perform a DFS and calculate the size and score of the current subtree
    private int dfs(int currentNode) {
        int subtreeSize = 1; // Current node's size is at least 1 (itself)
        long nodeScore = 1; // Initial score for the current node

        // Iterate through the children of the current node
        for (int childNode : childrenList.get(currentNode)) {
            int childSubTreeSize = dfs(childNode); // Recursively get child subtree size

            // Accumulate total subtree size and compute the score contribution of this child
            subtreeSize += childSubTreeSize;
            nodeScore *= childSubTreeSize;
        }

        // For non-root nodes, multiply the score with the size of the "rest of the tree"
        if (currentNode != 0) {
            nodeScore *= (nodeCount - subTreeSize);
        }

        // Compare and update the maximum score and the count of nodes
        if (nodeScore > maxNodeScore) {
            maxNodeScore = nodeScore;
            countMaxScoreNodes = 1;
        } else if (nodeScore == maxNodeScore) {
            countMaxScoreNodes++;
        }

        // Return the subtree size
        return subTreeSize;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <functional>
using namespace std;

class Solution {
public:
    // Member variables to keep track of the total count of nodes with the highest score,
    // the highest score itself, the current maximum score and the total number of nodes.
    int countMaxScoreNodes;
    long long maxNodeScore;
    int numNodes;

    // Function to start the process and return the number of nodes with the highest score
    int countHighestScoreNodes(vector<int>& parents) {
        numNodes = parents.size(); // Initialize the total number of nodes with maximum score to zero
        maxNodeScore = 0; // Initialize maximum score to zero
        countMaxScoreNodes = 0; // Set total number of nodes
        unordered_map<int, vector<int>> graph; // Create a graph from the parent array

        // Building the graph from the parent-child relationships
        for (int i = 1; i < numNodes; ++i) {
            graph[parents[i]].push_back(i);
        }

        // Start the DFS traversal from the root node, which is node 0
        dfs(0, graph);

        // Return the total count of nodes with the maximum score
        return countMaxScoreNodes;
    }

    // DFS function to calculate the score of each node and the size of the subtree rooted at each node
    int dfs(int currentNode, unordered_map<int, vector<int>>& graph) {
        int subtreeSize = 1; // Size of the subtree including the current node
        long long nodeScore = 1; // Product of the sizes of each subtree

        // For every child of the current node
        for (int childId : graph[nodeId]) {
            int childSubtreeSize = dfs(childId, graph); // Recursive DFS call
            subtreeSize += childSubtreeSize; // Update the size of the current subtree
            nodeScore *= childSubtreeSize; // Update the score by multiplying the sizes of subtrees
        }

        // If the node is not the root, multiply node score by the size of the "rest of the tree"
        if (nodeId != 0) {
            nodeScore *= (numNodes - subtreeSize);
        }

        // Update maxNodeScore and countOfMaximumNodes based on the calculated score for this node
        if (nodeScore > maxNodeScore) { // Found a new maximum score
            maxNodeScore = nodeScore; // Update maximum score
            countMaxScoreNodes = 1; // Reset count because we found a new maximum
        } else if (nodeScore == maxNodeScore) {
            ++countMaxScoreNodes; // Increment count because we found another node with the maximum score
        }

        // Return the total size of the subtree rooted at the current node
        return subtreeSize;
    }
};
```

Typescript Solution

```typescript
/**
 * This function counts the number of nodes in a tree that have the highest score.
 * The score of each node is calculated as the product of the sizes of its subtrees,
 * and if the node is not the root, then the size of the remaining tree is also considered.
 *
 * @param parents - an array where `parents[i]` is the parent of the `i`th node.
 * @returns the number of nodes with the highest score.
 */
function countHighestScoreNodes(parents: number[]): number {
    // The number of nodes in the tree
    const nodeCount = parents.length;

    // An adjacency list to store the tree, with each index representing a node and storing its children.
    let edges = Array.from({ length: nodeCount }, () => []);

    // Constructing the tree
    for (let i = 0; i < nodeCount; i++) {
        const parent = parents[i];
        if (parent !== -1) {
            edges[parent].push(i);
        }
    }

    // Initialize the count of nodes with the maximum score and the maximum score itself
    let maxScoreNodeCount = 0;
    let maxScore = 0;

    /**
     * A depth-first search function that calculates the size of the subtree rooted at this index
     * and the score for the current node.
     *
     * @param index - The current node index we are calculating size and score for
     * @returns The size of the subtree rooted at the given index
     */
    function dfs(index: number): number {
        let subtreeSize = 1;
        let nodeScore = 1;

        // Traversing children to calculate score and subtree sizes
        for (const childIndex of edges[index]) {
            const childSubtreeSize = dfs(childIndex);
            subtreeSize += childSubtreeSize;
            nodeScore *= childSubtreeSize;
        }

        // If the node is not root, the remaining tree size is also part of the score
        if (index > 0) {
            nodeScore *= nodeCount - subtreeSize;
        }

        // If the calculated score for the current node is higher than the recorded max, update it
        if (nodeScore > maxScore) {
            maxScore = nodeScore;
            maxScoreNodeCount = 1;
        } else if (nodeScore === maxScore) {
            // If the current node has a score equal to the max, increment the count
            maxScoreNodeCount++;
        }

        return subtreeSize;
    }

    // Start DFS from the root node (index 0)
    dfs(0);

    return maxScoreNodeCount;
}
```

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. The code uses a Depth-First Search (DFS) traversal to compute the size and score for each node exactly once. For each node, it looks into its children and calculates the score based on the size of the subtrees which are the results of the DFS calls. These operations all happen within the DFS function which is called once per node, leading to a linear time complexity with respect to the number of nodes.

Space Complexity

The space complexity of the code can also be considered $O(n)$ for several reasons:

- The adjacency list g, which represents the tree, will contain a maximum of $n-1$ edges, as it is a tree.
- The recursive DFS will at most be called recursively for a depth equal to the height of the tree. In the worst case (a skewed tree), the recursive call stack could also be $O(n)$.
- The auxiliary space required by the internal state of the DFS (variables like `size`, `score`, and the returned values) will not exceed $O(1)$ per call.

But, typically, the tree height is much less than n for a reasonably balanced tree, so the average case for the space complexity due to recursive calls is less than $O(n)$. However, since the worst-case scenario can still be $O(n)$ (for a skewed tree), we mention it as such.