339. Nested List Weight Sum

Medium Depth-First Search Breadth-First Search Leetcode Link

Problem Description

In this problem, we are given a `nestedList` consisting of integers that might be wrapped in multiple layers of lists. An integer's depth is determined by how many lists contain it. For instance, in the list `[1,[2,2],[[3],2],1]`, the number 1 is at depth 1 (since it's not within any nested list), the number 2 is at depth 2 (as it is in one nested list), and the number 3 is at depth 3 (as it's in two nested lists).

Our goal is to compute the sum of all integers in `nestedList` each weighted by its depth. In simple terms, we multiply each integer by the number of lists it is inside and then sum these products together.

Intuition

To arrive at the solution, we can utilize a depth-first search (DFS) strategy, where we traverse the nested list structure recursively. Whenever we encounter an integer, we multiply it by its depth and add it to the cumulative sum. If we encounter a nested list within this list, we call the DFS function recursively with the nested list and an incremented depth value.

The intuition behind using DFS is that it allows us to handle an arbitrary level of nesting in the list by recursively processing nested lists until we reach the most deeply nested integers. This method naturally handles the nested structure and allows us to apply the depth multiplier as we traverse the data.

Solution Approach

The solution uses a recursive function, `dfs`, which is a common algorithm in dealing with tree or graph data structures. In this case, the nested lists can be imagined as a tree where each list is a node, and its elements are either child nodes (other lists) or leaf nodes (integers).

The `dfs` function has two parameters, `nestedList`, which is the current list to be processed, and `depth`, which represents the current depth of traversal.

Here is a step-by-step breakdown of the `dfs` function:

1. Initialize a local variable `depth_sum` to `0`. This variable will hold the sum of the integers multiplied by their respective depths for the current list.
2. Iterate through each `item` in the `nestedList`:
   - If the item is an integer (checked using `item.isInteger()`), multiply the integer by its depth (`item.getInteger() * depth`) and add the result to `depth_sum`.
   - If the item is a nested list, make a recursive call to `dfs` with `item.getList()` and `depth + 1` to handle the deeper level, and add the result to `depth_sum`.
3. After iterating through all items, return `depth_sum` to the caller.

The function kicks off by calling `dfs` with the initial `nestedList` and a starting depth of `1`. As `dfs` progresses through each level of nesting, it adds to the cumulative sum while correctly adjusting the depth.

Effectively, this approach applies a depth-first traversal on the tree structure of nested lists, calculating the product of the integer values and their respective depths at each node and aggregating these values up the call stack.

Here is the main function call:

```
return dfs(nestedList, 1)
```

It starts the recursive process and eventually returns the required weighted sum after the entire nested structure has been explored.

Example Walkthrough

Let's explain the depth-first search (DFS) approach with a small example, consider the nested list `[1, [2, 2], [[3], 2], 1]`.

1. The DFS process starts by calling `dfs(nestedList, 1)` with the initial depth as `1`.
2. Starting the DFS algorithm, the function initializes a local variable `depth_sum` to `0`.
3. The function begins to iterate over each item in the `nestedList`.
4. The first element `1` is an integer and not a nested list.
   - Since it's at depth `1`, we multiply it by its depth: `1 * 1`.
   - We add this to `depth_sum`, which is `0 + 1 = 1`.
5. The second element `[2, 2]` is a nested list, so we call `dfs([2, 2], 2)` because we are now one layer deeper.
   - This call itself will add `2 * 2 + 2 * 2` to our cumulative sum because both `2`s are at depth `2`.
   - After this recursive call, `depth_sum` is updated by `1 + 8 = 9`.
6. The third element `[[3], 2]` is another nested list. We now encounter two cases:
   - The nested list `[3]` requires a recursive call: `dfs([3], 3)`.
     - The element `3` at depth `3` gives `3 * 3`, and the `depth_sum` is increased by `9`.
   - The integer `2` is at depth `2`, resulting in `2 * 2`.
   - Combining these, we add `9 + 4 = 13` to our `depth_sum`, resulting in `9 + 13 = 22`.
7. Finally, the last element `1` is an integer.
   - It's at depth `1`, so it contributes `1 * 1` to our sum.
   - The final `depth_sum` is updated to `22 + 1 = 23`.

Therefore, the result of the DFS algorithm when applied to our example nested list `[1, [2, 2], [[3], 2], 1]` yields `23`. This is how the recursive function cumulatively builds up the sum of the products of integers and their respective depths as it iterates through the nested list.

Python Solution

```python
# Assuming NestedInteger class definition exists (as provided in the problem description)

class Solution:
    def depth_sum(self, nested_list):
        """
        Calculate the sum of all integers in the nested list weighted by their depth.

        :param nested_list: List[NestedInteger] - a list of NestedInteger
        :return: int - depth sum of input nested list
        """

        # Helper recursive function to calculate the depth sum
        def dfs(current_list, current_depth):
            current_depth_sum = 0  # Initialize the depth sum for the current level

            # Loop through each item in the current nested list
            for item in current_list:
                if item.isInteger():
                    # If the item is an integer, add its value times the current depth
                    current_depth_sum += item.getInteger() * current_depth
                else:
                    # If the item is a list, recursively call dfs to calculate its depth sum
                    current_depth_sum += dfs(item.getList(), current_depth + 1)

            return current_depth_sum  # Return the calculated depth sum for this level

        return dfs(nested_list, 1)  # Start the depth-first search with the top-level list and depth 1

# The code above would be used as part of a larger solution where the NestedInteger class is defined.

```

Java Solution

```java
class Solution {

    // Calculate the sum of all integers within the nested list,
    // where each integer is multiplied by its depth in the nested list structure.
    public int depthSum(List<NestedInteger> nestedList) {
        return computeDepthSum(nestedList, 1); // Starting with depth level 1
    }

    // Recursive function to compute the depth sum
    private int computeDepthSum(List<NestedInteger> nestedList, int depth) {
        int totalSum = 0; // Initialize sum as 0

        // Iterate over each element in the nested list
        for (NestedInteger item : nestedList) {
            // Check if item is a single integer
            if (item.isInteger()) {
                // If it's an integer, add its value multiplied by its depth level to totalSum
                totalSum += item.getInteger() * depth;
            } else {
                // Otherwise, perform a recursive call on the sublist with increased depth
                totalSum += computeDepthSum(item.getList(), depth + 1);
            }
        }

        // Return the computed sum
        return totalSum;
    }
}

```

C++ Solution

```cpp
#include <vector>

// Forward declaration of the interface that will be used in the functions.
class NestedInteger {
public:
    // Return true if this NestedInteger holds a single integer, rather than a nested list.
    bool isInteger() const;

    // Return the single integer that this NestedInteger holds, if it holds a single integer.
    // The result is undefined if this NestedInteger holds a nested list.
    int getInteger() const;

    // Return the nested list that this NestedInteger holds, if it holds a nested list.
    // The result is undefined if this NestedInteger holds a single integer.
    const std::vector<NestedInteger> &getList() const;
};

/**
 * This function calculates the sum of values in a nested list structure,
 * where each element is multiplied by its depth in the nested structure.
 *
 * @param nestedList A list of NestedInteger.
 * @return The depth sum of the nested list.
 */
int depthSum(const std::vector<NestedInteger>& nestedList) {
    // Helper function to perform a depth-first search on the nested list.
    std::function<int(const std::vector<NestedInteger>&, int)> dfs = [&](const std::vector<NestedInteger>& list, int depth) -> int {
        int currentDepthSum = 0;

        // Iterate through each element in the nested list.
        for (const auto& item : list) {
            if (item.isInteger()) {  // Check if the item is an integer.
                // Item is an integer, so add its value multiplied by its depth to the sum.
                currentDepthSum += item.getInteger() * depth;
            } else {
                // Item is not an integer (it's a nested list), so call dfs recursively.
                currentDepthSum += dfs(item.getList(), depth + 1);
            }
        }

        // Return the sum for the current depth.
        return currentDepthSum;
    };

    // Call the dfs helper function with the initial depth of 1 for the outermost list.
    return dfs(nestedList, 1);
}

```

Typescript Solution

```typescript
/**
 * This TypeScript function calculates the sum of values in a nested list structure,
 * where each element is multiplied by its depth in the nested structure.
 *
 * @param {NestedInteger[]} nestedList - A list of NestedInteger (definitions provided by the interface but not implemented here).
 * @returns {number} - The depth sum of the nested list.
 */
function depthSum(nestedList: NestedInteger[]): number {
    /**
     * Helper function to perform a depth-first search on the nested list.
     *
     * @param {NestedInteger[]} list - A nested list to be processed.
     * @param {number} depth - The depth level of the current nested list.
     * @returns {number} - The calculated depth sum at the current level.
     */
    const dfs = (list: NestedInteger[], depth: number): number => {
        let currentDepthSum = 0;

        // Iterate through each element in the nested list.
        for (const item of list) {
            if (item.isInteger()) {  // Check if the item is an integer.
                // Item is an integer, so add its value multiplied by its depth to the sum.
                currentDepthSum += item.getInteger() * depth;
            } else {
                // Item is not an integer (it is a nested list), so call dfs recursively.
                currentDepthSum += dfs(item.getList(), depth + 1);
            }
        }

        // Return the sum for the current depth.
        return currentDepthSum;
    };

    // Call the dfs helper function with the initial depth of 1 for the outermost list.
    return dfs(nestedList, 1);
}

```

Time and Space Complexity

The given code defines a function `dfs` that traverses a nested list structure to compute the weighted sum of all the integers within, where each integer is multiplied by its depth level.

Time Complexity

The time complexity of the function is `O(N)`, where `N` is the total number of integers and lists within all levels of the nested list. Specifically, the function `dfs` visits each element exactly once. For each integer it encounters, it performs a constant time operation of multiplication and addition. For each list, it makes a recursive call to process its elements. However, since every element is only visited once, the overall time to visit all elements is proportional to their count.

Space Complexity

The space complexity of the function is `O(D)`, where `D` is the maximum depth of the nested list. This complexity arises from the call stack used for recursion. In the worst case, the recursion will go as deep as the deepest nested list. Therefore, the maximum number of nested calls will equal the maximum depth `D`. Furthermore, there is only a constant amount of space used at each level for variables such as `depth_sum`.