364. Nested List Weight Sum II

Medium Array Depth-First Search Breadth-First Search

Problem Description

This LeetCode problem involves calculating the weighted sum of integers within a nested list, where the weight depends on the depth of each integer in the nested structure. The nested list is a list that can contain integers as well as other nested lists to any depth. The depth of an integer is defined by how many lists are above it. For example, if we have the nested list `[1,[2,2],[[3],2],1]`, the integer 1 at the start and end is at depth 1, the integers 2 in the first inner list are at depth 2, the integer 3 is at depth 3 since it is inside two lists, and so on.

To determine the weighted sum, we need to calculate the 'weight' of each integer, which is defined as the maximum depth of any integer in the entire nested structure minus the depth of that integer, plus 1. The task then is to calculate this weighted sum of all integers in the nested list.

Intuition

To arrive at the solution for this problem, we need to follow a two-step approach.

First, we determine the maximum depth of the nested list. This requires us to traverse the nested lists and keep track of the depth as we go. We can do this using a `max_depth` helper function that recursively goes through each element, increasing the depth when it encounters a nested list and comparing the current depth with the maximum depth found so far.

Second, we calculate the weighted sum of all integers within the nested list structure using this maximum depth. We can create a `dfs` (depth-first search) helper function that recursively traverses the nested list structure. When the function encounters an integer, it multiplies it by the weight, which is the maximum depth minus the current depth of the integer plus one. If it encounters another nested list, it calls itself with the new depth that's one less than the current maximum depth. By summing up the results of these multiplications and recursive calls, we get the weighted sum of all integers in the nested list.

Solution Approach

The implementation of the solution utilizes a recursive depth-first search (DFS) approach. This is a common pattern for traversing tree or graph-like data structures, which is similar to the nested list structure we're dealing with here. The Solution class provides two functions: `max_depth` and `dfs`, which work together to solve the problem.

`max_depth` is a helper function that calculates the deepest level of nesting in the given `nestedList`. It initializes a variable `depth` to 1, to represent the depth of the outermost list, and iterates through each item in the current list. For each item that is not an integer (i.e., another nested list), it makes a recursive call to get the maximum depth of that list and compares it with the current `depth` to keep track of the highest value. The function returns the maximum depth it finds.

The `dfs` function is the core of the depth-first search algorithm. It operates recursively, computing the sum of the integers in the `nestedList`, weighted by their depth. For each item in `nestedList`, it checks whether the item is an integer or a nested list. If it's an integer, the function calculates the weight of the integer using the formula `maxDepth - (the depth of the integer) + 1` and adds this weighted integer to the `depth_sum`. If the item is a nested list, the `dfs` function is called recursively, with `max_depth` decreased by 1 to account for the increased nesting. This ensures that integers nested deeper inside the structure are weighted appropriately. The computed `depth_sum` for each recursive call is then added up to form the total sum.

Finally, the `depthSumInverse` function of the `Solution` class uses these helper functions to calculate and return the final weighted sum. It first calls `max_depth` to find the maximum depth of the list, and then calls `dfs`, passing the `nestedList` and the maximum depth as arguments.

Altogether, this is an efficient and elegant solution that makes clever use of recursion to traverse and process the nested list structure.

Example Walkthrough

Let's use a small nested list example to illustrate the solution approach: `[2, [1, [3]], 4]`.

1. **Calculating Maximum Depth**:
   - We start by determining the maximum depth of the nested structure with the `max_depth` function.
   - The list `[2, [1, [3]], 4]` starts with depth 1 at the outermost level.
   - Element '2' is an integer at depth 1.
   - Element '[1, [3]]' is a nested list. We apply `max_depth` recursively.
     - Inside this list, '1' is an integer at depth 2.
     - The element '[3]' is a nested list. Again, we apply `max_depth` recursively.
       - The integer '3' is at depth 3.
   - Element '4' is an integer at depth 1.
   - The maximum depth of these elements is 3. This is the `maxDepth`.
2. **Calculating Weighted Sum via DFS**:
   - Next, we use the `dfs` function to calculate the weighted sum.
   - For each element, if it's an integer, we calculate its weighted value by `maxDepth - currentDepth + 1`.
   - Start with element '2', which is an integer at depth 1. Its weight is `3 - 1 + 1 = 3`. So it contributes `2 * 3 = 6` to the sum.
   - Move to element '[1, [3]]', which is a nested list. We apply `dfs` recursively.
     - The integer '1' at depth 2 has a weight of `3 - 2 + 1 = 2`. It contributes `1 * 2 = 2`.
     - For the nested list '[3]', apply `dfs` recursively.
       - The integer '3' at depth 3 has a weight of `3 - 3 + 1 = 1`. It contributes `3 * 1 = 3`.
   - Finally, the integer '4' at depth 1 has a weight of `3 - 1 + 1 = 3`. It contributes `4 * 3 = 12`.
   - The sum of all weighted integers is `6 + 2 + 3 + 12 = 23`.
3. **Combining the Functions (The `depthSumInverse` Function)**:
   - The `depthSumInverse` function combines the use of both `max_depth` and `dfs`.
   - First, it finds the maximum depth with `max_depth(nestedList)` which gives us `maxDepth = 3`.
   - Then it calculates the weighted sum with `dfs(nestedList, maxDepth)`, which gives us the weighted sum `23`.
   - It returns `23` as the final result.

To summarize, this approach efficiently processes each integer with its appropriate weight, determined by its depth in the nested list structure, to calculate the requested weighted sum.

Solution Implementation

Python

```python
# Using the interface provided, we define a solution class with methods to calculate the inverse depth sum for a nested integer list
class Solution:
    def depthSumInverse(self, nested_list):
        # This helper function computes the maximum depth of the nested integer list.
        def max_depth(nested_list):
            depth = 1  # Start with depth 1 since there's at least one level of nesting.
            for item in nested_list:
                if not item.isInteger():
                    # For each nested list, calculate the depth recursively and update the maximum depth accordingly.
                    current_depth = max_depth(item.getList()) + 1
                    depth = max(depth, current_depth)
            return depth  # Return the maximum depth found.

        # This is a helper function that computes the sum of integers in the nested list, each multiplied by its "inverse" depth.
        def dfs(nested_list, level_multiplier):
            depth_sum = 0  # Initialize depth sum.
            for item in nested_list:
                if item.isInteger():
                    # If the item is an integer, multiply it by the level_multiplier.
                    depth_sum += item.getInteger() * level_multiplier
                else:
                    # If the item is a list, make a recursive call and decrease the level_multiplier.
                    depth_sum += dfs(item.getList(), level_multiplier - 1)
            return depth_sum  # Return the computed depth sum for this level.

        max_depth = max_depth(nested_list)  # Calculate the maximum depth of the input nested list.
        return dfs(nested_list, max_depth_value)  # Call the dfs function starting with the maximum depth as the level multiplier.
```

Java

```java
class Solution {
    // Calculate the inverse depth sum of the given nest integer list.
    public int depthSumInverse(List<NestedInteger> nestedList) {
        // First, find the maximum depth of the nested list.
        int maxDepth = findMaxDepth(nestedList);
        // Then, calculate the depth sum with depth weights in inverse order.
        return calculateDepthSumInverse(nestedList, maxDepth);
    }

    // A helper method to determine the maximum depth of the nested integer list.
    private int findMaxDepth(List<NestedInteger> nestedList) {
        int depth = 1; // Initialize the minimum depth.
        for (NestedInteger item : nestedList) {
            // If the current item is a list, calculate its depth.
            if (!item.isInteger()) {
                // Recursively find the max depth for the current list + 1 for the current level.
                depth = Math.max(depth, 1 + findMaxDepth(item.getList()));
            }
            // If it's an integer, it does not contribute to increasing the depth.
        }
        return depth; // Return the maximum depth found.
    }

    // A helper method to recursively calculate the weighted sum of integers at each depth.
    private int calculateDepthSumInverse(List<NestedInteger> nestedList, int weight) {
        int depthSum = 0; // Initialize the sum for the current level.
        for (NestedInteger item : nestedList) {
            // If the current item is an integer, multiply it by its depth weight.
            if (item.isInteger()) {
                depthSum += item.getInteger() * weight;
            } else {
                // If the item is a list, recursively calculate the sum of its elements
                // with the weight reduced by 1 since we're going one level deeper.
                depthSum += calculateDepthSumInverse(item.getList(), weight - 1);
            }
        }
        return depthSum; // Return the total sum for this level.
    }
}
```

C++

```cpp
#include <vector>
#include <algorithm>

// Forward declaration of the NestedInteger class interface. Assuming it is predefined.
class NestedInteger {
public:
    // Returns true if this NestedInteger holds a single integer, rather than a nested list.
    bool isInteger() const;

    // Returns 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;

    // Returns 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;
};

/**
 * Calculates the maximum depth of nested lists within a list of NestedInteger
 * @param nested_list - The list of NestedInteger to evaluate.
 * @return The maximum depth found.
 */
int getMaxDepth(const std::vector<NestedInteger>& nested_list) {
    int depth = 1;
    for (const auto& item : nested_list) {
        if (!item.isInteger()) {
            // Recursively find the depth of the nested list and compare it with the current depth
            depth = std::max(depth, 1 + getMaxDepth(item.getList()));
        }
    }
    return depth;
}

/**
 * Recursively calculates the inverse depth sum of a NestedInteger list
 * @param nested_list - The current level of NestedInteger
 * @param depth - The depth to multiply the integers with
 * @return The calculated depth inverse sum.
 */
int calculateDepthInverseSum(const std::vector<NestedInteger>& nested_list, int depth) {
    int depth_sum = 0;
    for (const auto& item : nested_list) {
        if (item.isInteger()) {
            // Multiply the integer by its depth
            depth_sum += item.getInteger() * depth;
        } else {
            // Recursively calculate the depth sum of nested lists, one level deeper
            depth_sum += calculateDepthInverseSum(item.getList(), depth - 1);
        }
    }
    return depth_sum;
}

/**
 * Calculates the sum of all integers in a NestedInteger list weighted by their depth,
 * with the integers deepest in the list weighted the most.
 * @param nested_list - The list of NestedInteger to sum up
 * @return The weighted sum where deeper integers carry more weight.
 */
int depthSumInverse(const std::vector<NestedInteger>& nested_list) {
    // Find the maximum depth to start the inverse weighting
    int max_depth = getMaxDepth(nested_list);
    // Calculate the inverse depth sum starting from the maximum depth
    return calculateDepthInverseSum(nested_list, max_depth);
}
```

TypeScript

```typescript
// This TypeScript function calculates the inverse depth sum of a NestedInteger list.

/**
 * Calculates the maximum depth of nested lists within a list of NestedInteger
 * @param {NestedInteger[]} nestedList - The list of NestedInteger to evaluate.
 * @return {number} - The maximum depth found.
 */
function getMaxDepth(nestedList: NestedInteger[]): number {
    let depth: number = 1;
    for (const item of nestedList) {
        if (!item.isInteger()) {
            // Recursively find the depth of this nested list and compare it with the current depth
            depth = Math.max(depth, 1 + getMaxDepth(item.getList()));
        }
    }
    return depth;
}

/**
 * Recursively calculates the inverse depth sum of a NestedInteger list
 * @param {NestedInteger[]} nestedList - The current level of NestedInteger
 * @param {number} depth - The depth to multiply the integers with
 * @return {number} - The calculated depth inverse sum.
 */
function calculateDepthInverseSum(nestedList: NestedInteger[], depth: number): number {
    let depthSum: number = 0;
    for (const item of nestedList) {
        if (item.isInteger()) {
            // Multiply the integer by its depth
            depthSum += item.getInteger() * depth;
        } else {
            // Recursively calculate the depth sum of nested lists, one level deeper
            depthSum += calculateDepthInverseSum(item.getList(), depth - 1);
        }
    }
    return depthSum;
}

/**
 * Calculates the sum of all integers in a NestedInteger list weighted by their depth,
 * with the integers deepest in the list weighted the most.
 * @param {NestedInteger[]} nestedList - The list of NestedInteger to sum up
 * @return {number} - The weighted sum where deeper integers carry more weight.
 */
function depthSumInverse(nestedList: NestedInteger[]): number {
    // Find the maximum depth to start the inverse weighting
    const depth: number = getMaxDepth(nestedList);
    // Calculate the inverse depth sum starting from the maximum depth
    return calculateDepthInverseSum(nestedList, depth);
}
```

Time and Space Complexity

Time Complexity

The time complexity of the provided code is primarily dependent on two functions: `max_depth` and `dfs`.

- The `max_depth` function computes the maximum depth of the nested list. In the worst case, it visits each element in the nested list and calculates the depth by making a recursive call for each list it encounters. This results in a time complexity of `O(N)`, where `N` is the total number of elements and nested lists within the outermost list because it has to potentially go through all elements at different levels of nesting to calculate the maximum depth.
- The `dfs` (depth-first search) function visits each element in the nested list once and for each integer it finds, it performs an operation that takes `O(1)` time. Where the function encounters nested lists, it makes a recursive call, decrementing the `max_depth` by one. The time complexity of `dfs` is also `O(N)`, with the same definition of `N` as above.

Therefore, the overall time complexity of the code is `O(N)`, combining the time it takes to calculate the maximum depth and then to perform the depth-first search.

Space Complexity

The space complexity is taken up by the recursion call stack in both `max_depth` and `dfs` functions.

- The `max_depth` function will occupy space on the call stack up to the maximum depth of `D`, where `D` is the maximum level of nesting, resulting in a space complexity of `O(D)`.
- The `dfs` function also uses recursion, and in the worst-case scenario, it will have a stack depth of `D` as well, giving us another `O(D)`.

Since these functions are not called recursively within each other—but instead, one after the other—the overall space complexity does not multiply, and the space complexity remains `O(D)`.

In conclusion, the time complexity of the code is `O(N)` and the space complexity is `O(D)`.