<https://leetcode.ca/all/364.html>

Given a nested list of integers, return the sum of all integers in the list weighted by their depth.

Each element is either an integer, or a list -- whose elements may also be integers or other lists.

Different from the [previous question](https://leetcode.com/problems/nested-list-weight-sum/) where weight is increasing from root to leaf, now the weight is defined from bottom up. i.e., the leaf level integers have weight 1, and the root level integers have the largest weight.

**Example 1:**

**Input:** [[1,1],2,[1,1]]

**Output:** 8

**Explanation:** Four 1's at depth 1, one 2 at depth 2.

**Example 2:**

**Input:** [1,[4,[6]]]

**Output:** 17

**Explanation:** One 1 at depth 3, one 4 at depth 2, and one 6 at depth 1; 1\*3 + 4\*2 + 6\*1 = 17.

/\*\*

\* // This is the interface that allows for creating nested lists.

\* // You should not implement it, or speculate about its implementation

\* public interface NestedInteger {

\* // Constructor initializes an empty nested list.

\* public NestedInteger();

\*

\* // Constructor initializes a single integer.

\* public NestedInteger(int value);

\*

\* // @return true if this NestedInteger holds a single integer, rather than a nested list.

\* public boolean isInteger();

\*

\* // @return the single integer that this NestedInteger holds, if it holds a single integer

\* // Return null if this NestedInteger holds a nested list

\* public Integer getInteger();

\*

\* // Set this NestedInteger to hold a single integer.

\* public void setInteger(int value);

\*

\* // Set this NestedInteger to hold a nested list and adds a nested integer to it.

\* public void add(NestedInteger ni);

\*

\* // @return the nested list that this NestedInteger holds, if it holds a nested list

\* // Return null if this NestedInteger holds a single integer

\* public List<NestedInteger> getList();

\* }

\*/

**Attempt 1: 2024-06-10**

**Solution 1: DFS (30 min)**

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 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.

}

}

**Refer to**

<https://algo.monster/liteproblems/364>

**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 (weight 3), the integers 2 in the first inner list are at depth 2 (weight 2), the integer 3 is at depth 3 (weight 1) 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.

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 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.

}

}

**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).

**Refer to**

<https://github.com/cheonhyangzhang/leetcode-solutions/blob/master/364-nested-list-weight-sum-ii.md>

**DFS**

public class Solution {

public int depthSumInverse(List<NestedInteger> nestedList) {

int height = getHeight(nestedList);

return ds(nestedList, height);

}

private int getHeight(List<NestedInteger> nestedList) {

int height = 1;

for (NestedInteger ni:nestedList) {

if (!ni.isInteger()) {

int tmp = getHeight(ni.getList());

height = Math.max(height, tmp + 1);

}

}

return height;

}

private int ds(List<NestedInteger> nestedList, int level) {

int sum = 0;

for (NestedInteger ni:nestedList) {

if (ni.isInteger()) {

sum += level \* ni.getInteger();

}

else {

sum += ds(ni.getList(), level - 1);

}

}

return sum;

}

}

**BFS**

class Solution {

public int depthSumInverse(List<NestedInteger> nestedList) {

Queue<List<NestedInteger>> q = new LinkedList<List<NestedInteger>>();

q.add(nestedList);

Stack<Integer> stack = new Stack<Integer>();

while (!q.isEmpty()) {

int size = q.size();

int sum = 0;

for (int i = 0; i < size; i ++) {

List<NestedInteger> node = q.poll();

for (NestedInteger ni : node) {

if (ni.isInteger()) {

sum += ni.getInteger();

} else {

q.add(ni.getList());

}

}

}

stack.push(sum);

}

int level = 1;

int res = 0;

while (!stack.isEmpty()) {

res += level \* stack.pop();

level ++;

}

return res;

}

}

**Refer to**

<https://www.cnblogs.com/grandyang/p/5615583.html>

这道题是之前那道 [Nested List Weight Sum](http://www.cnblogs.com/grandyang/p/5340305.html) 的拓展，与其不同的是，这里的深度越深，权重越小，和之前刚好相反。但是解题思路没有变，还可以用 DFS 来做

**Refer to Deepseek**

The Single-Pass Solution for LeetCode 364, "Nested List Weight Sum II," avoids the need for two separate traversals (one to find the maximum depth and another to calculate the weighted sum). Instead, it uses a \*\*BFS-like approach\*\* to accumulate the sum level by level and then calculates the weighted sum in a single pass.

**Approach**

Level-by-Level Traversal:

Use a queue to process the nested list level by level.

At each level, keep track of the sum of integers at that level.

After processing all levels, calculate the weighted sum by multiplying each level's sum by its corresponding weight (from bottom to top).

Weight Calculation:

The weight of a level is determined by its distance from the bottom. For example:

The bottom level (leaf nodes) has a weight of 1.

The level above it has a weight of 2, and so on.

Accumulate Level Sums:

Use a list to store the sum of integers at each level.

After processing all levels, iterate through the list in reverse order to calculate the weighted sum.

class Solution {

public int depthSumInverse(List<NestedInteger> nestedList) {

// List to store the sum of integers at each level

List<Integer> levelSums = new ArrayList<>();

// Queue for BFS traversal

Queue<NestedInteger> queue = new LinkedList<>(nestedList);

// Perform BFS

while (!queue.isEmpty()) {

int levelSize = queue.size();

int levelSum = 0;

// Process all elements at the current level

for (int i = 0; i < levelSize; i++) {

NestedInteger ni = queue.poll();

if (ni.isInteger()) {

// Add the integer to the current level sum

levelSum += ni.getInteger();

} else {

// Add all elements of the nested list to the queue

queue.addAll(ni.getList());

}

}

// Add the current level sum to the list

levelSums.add(levelSum);

}

// Calculate the weighted sum

int weightedSum = 0;

int depth = levelSums.size(); // Total number of levels

for (int i = 0; i < depth; i++) {

// Weight = depth - i (bottom-up weight)

weightedSum += levelSums.get(i) \* (depth - i);

}

return weightedSum;

}

}

**Explanation of the Code**

Initialization:

A levelSums list is used to store the sum of integers at each level.

A queue is initialized with the input nestedList for BFS traversal.

BFS Traversal:

For each level, process all elements in the queue:

If the element is an integer, add it to the current level's sum.

If the element is a list, add all its elements to the queue for processing in the next level.

After processing a level, add its sum to the levelSums list.

Weighted Sum Calculation:

After processing all levels, calculate the weighted sum by iterating through the levelSums list in reverse order.

The weight for each level is (depth - i), where depth is the total number of levels and i is the current index.

**Example Walkthrough**

**Example 1:**

Input: [[1,1],2,[1,1]]

Execution:

Level 1:

Elements: [[1,1], 2, [1,1]]

Sum: 2 (only the integer 2 is at level 1).

Queue after processing: [1,1,1,1] (from the nested lists [1,1] and [1,1]).

Level 2:

Elements: [1,1,1,1]

Sum: 1 + 1 + 1 + 1 = 4.

Queue after processing: Empty.

Level Sums:

levelSums = [2, 4]

Weighted Sum:

Depth = 2.

Weight for Level 1: 2 - 0 = 2 → 2 \* 2 = 4.

Weight for Level 2: 2 - 1 = 1 → 4 \* 1 = 4.

Total weighted sum: 4 + 4 = 8.

Output: 8

**Example 2:**

Input: [1,[4,[6]]]

Execution:

Level 1:

Elements: [1, [4,[6]]]

Sum: 1 (only the integer 1 is at level 1).

Queue after processing: [4,[6]].

Level 2:

Elements: [4,[6]]

Sum: 4 (only the integer 4 is at level 2).

Queue after processing: [6].

Level 3:

Elements: [6]

Sum: 6.

Queue after processing: Empty.

Level Sums:

levelSums = [1, 4, 6]

Weighted Sum:

Depth = 3.

Weight for Level 1: 3 - 0 = 3 → 1 \* 3 = 3.

Weight for Level 2: 3 - 1 = 2 → 4 \* 2 = 8.

Weight for Level 3: 3 - 2 = 1 → 6 \* 1 = 6.

Total weighted sum: 3 + 8 + 6 = 17.

Output: 17

**Complexity Analysis**

Time Complexity:

Every element in the nested list is processed once during BFS traversal.

If there are N total elements, the time complexity is O(N).

Space Complexity:

The queue and levelSums list use O(N) space in the worst case.

This single-pass solution is efficient and avoids the overhead of a second traversal, making it a better choice for large inputs.

**Refer to**

[L104.Maximum Depth of Binary Tree (Ref.L222,L366)](note://DFEAE5072F224B7F85934A11F3381C70)

[L339.Nested List Weight Sum (Ref.L341,L364)](note://WEB29fa11663191cf451aba40f451db5294)