<https://leetcode.ca/all/339.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.

**Example 1:**

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

**Output:** 10

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

**Example 2:**

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

**Output:** 27

**Explanation:** One 1 at depth 1, one 4 at depth 2, and one 6 at depth 3; 1 + 4\*2 + 6\*3 = 27.

/\*\*

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

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

\* public interface NestedInteger {

\*

\* // @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();

\*

\* // @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: 2024-05-27**

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

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;

}

}

**Refer to**

<https://www.cnblogs.com/apanda009/p/7948211.html>

 fb: 原题的weight是1,2,3,...

改成了1,2,6,24,...这样子

Time Complexity: O(n). n 是指全部叶子的数目加上dfs走过层数的总数. [[[[[5]]]],[[3]], 1], 3个叶子, dfs一共走了6层. 所以用了 3 + 6 = 9 的时间.

Space: O(D). D 是recursive call用的stack的最大数目, 即是最深的层数, 上面例子最深走过4层, 这里D = 4.

**Recursive (DFS)**

Depth and unpacked nested list are being passed to the next recursive call. Anytime, an integer is encountered, a value is evaluated and incremented to the stack's sum. The sum is then returned to the caller once the for loop ends. Since this is a tree, there is no need to keep track of visited nodes like general graph traversal.

public class Solution {

public int depthSum(List<NestedInteger> nestedList) {

return depthSum(nestedList, 1);

}

public int depthSum(List<NestedInteger> nestedList, int weight) {

int sum = 0;

for (NestedInteger each : nestedList) {

if (each.isInteger())

sum += each.getInteger() \* weight;

else sum += depthSum(each.getList(), weight+1);

}

return sum;

}

}

**Step by Step (with issue)**

**注意，以下是一个博主的个人实现，经过实际测试，该实现在运行其他网站的标准答案的时候出现了异常，同样的输入{1, {4, {6}}}，在运行别的网站的类似答案的时候都是输出18，而不是27，只有在运行博主自己的 NestedIntegerSum class 的逻辑的时候才会输出27，最大的区别在于博主的答案里面以下部分并不是以 if - else 的逻辑区分开来，而是每一层都先计算了 current.getInteger() \* level 并且加入到 tempSum 中，这样在后面 for 循环递归更深的层回来之后 tempSum 不会丢失当前层的 current.getInteger() \* level 值，但是在其他网站的答案中 if - else 逻辑是严格区分开的，并没有在每一层累加 current.getInteger() \* level 的逻辑，导致要是以博主自己实现的结构运行的时候递归更深的层回来之后 tempSum 会丢失当前层的 current.getInteger() \* level 值，应该是博主自己实现的数据结构有问题**

int tempSum = 0;

if (current.getInteger() != null) {

tempSum = current.getInteger() \* level;

}

for (NestedInteger nestedCurrent : current.getList()) {

tempSum += depthSumNestedInteger(nestedCurrent, level + 1);

}

return tempSum;

/\*\*

\* This is the interface that represents nested lists. You should not implement

\* it, or speculate about its implementation.

\*/

interface NestedInteger {

/\*\*

\* @return true if this NestedInteger holds a single integer, rather than a

\* nested list

\*/

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

\*/

Integer getInteger();

/\*\*

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

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

\* integer

\*/

List<NestedInteger> getList();

}

class MyNestedIneteger implements NestedInteger {

private Integer theSingleInteger;

private List<NestedInteger> theList;

public MyNestedIneteger(Integer theSingleInteger, List<NestedInteger> theList) {

this.theSingleInteger = theSingleInteger;

this.theList = theList;

}

@Override

public boolean isInteger() {

return null == theList && null != theSingleInteger;

}

@Override

public Integer getInteger() {

return theSingleInteger;

}

@Override

public List<NestedInteger> getList() {

return theList;

}

@Override

public String toString() {

StringBuilder string = new StringBuilder();

string.append("{");

if (null != theSingleInteger) {

string.append(theSingleInteger);

}

if (null != theList) {

for (NestedInteger current : theList) {

string.append(", " + current.toString());

}

}

string.append("}");

return string.toString();

}

}

public class NestedIntegerSum {

/\*\*

\* Given a nested list of integers, returns the sum of all integers in the

\* list weighted by their depth For example, given the list {{1,1},2,{1,1}}

\* the function should return 10 (four 1's at depth 2, one 2 at depth 1)

\* Given the list {1,{4,{6}}} the function should return 27 (one 1 at depth

\* 1, one 4 at depth 2, and one 6 at depth 3)

\*/

public static int depthSum(List<NestedInteger> input) {

if (null == input) {

return 0;

} else {

int sum = 0;

for (NestedInteger current : input) {

sum += depthSumNestedInteger(current, 1);

}

return sum;

}

}

private static int depthSumNestedInteger(NestedInteger current, int level) {

System.out.println("current: " + current + " level: " + level);

if (null == current) {

return 0;

} else {

if (current.isInteger()) {

return current.getInteger() \* level;

} else {

int tempSum = 0;

if (current.getInteger() != null) {

tempSum = current.getInteger() \* level;

}

for (NestedInteger nestedCurrent : current.getList()) {

tempSum += depthSumNestedInteger(nestedCurrent, level + 1);

}

return tempSum;

}

}

}

public static void main(String[] args) {

List<NestedInteger> list1 = new ArrayList<>();

List<NestedInteger> list2 = new ArrayList<>();

NestedInteger nestedInteger1 = new MyNestedIneteger(1, list1);

NestedInteger nestedInteger2 = new MyNestedIneteger(4, list2);

NestedInteger nestedInteger3 = new MyNestedIneteger(6, null);

list1.add(nestedInteger2);

list2.add(nestedInteger3);

List<NestedInteger> input = new ArrayList<>();

input.add(nestedInteger1);

System.out.println(input);

System.out.println(depthSum(input));

}

}

**Iterative (BFS)**

This can also be categorized as Level-order tree traversal algorithm. The algorithm will travel level by level and increment the running sum when integer is encountered. Queue is used as data structure due to the nature of BFS where it needs to process data sequentially for each level.

public int depthSum(List<NestedInteger> nestedList) {

if(nestedList == null){

return 0;

}

int sum = 0;

int level = 1;

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

while(queue.size() > 0){

int size = queue.size();

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

NestedInteger ni = queue.poll();

if(ni.isInteger()){

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

}else{

queue.addAll(ni.getList());

}

}

level++;

}

return sum;

}

**Refer to**

<https://algo.monster/liteproblems/339>

**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:

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.

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.

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].

The DFS process starts by calling dfs(nestedList, 1) with the initial depth as 1.

Starting the DFS algorithm, the function initializes a local variable depth\_sum to 0.

The function begins to iterate over each item in the nestedList.

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.

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 2s are at depth 2.

After this recursive call, depth\_sum is updated by 1 + 8 = 9.

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.

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.

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;

}

}

**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.

**Refer to**

[L364.Nested List Weight Sum II (Ref.L339)](note://WEB591a4d7a957ba5748d1602d3fcc4b096)