2649. Nested Array Generator

Medium

Leetcode Link

Problem Description

The problem requires us to work with a multi-dimensional array structure. This is an array where each element can either be an integer or another array of similar structure (which again could contain integers or other arrays). Our objective is to create a generator function that traverses this nested array structure in an 'inorder' manner. In the context of this problem, 'inorder traversal' means sequentially iterating over each element in the array. If the element is an integer, it is yielded by the generator. If the element is a sub-array, the generator applies the same inorder traversal to it.

Intuition

The solution requires an understanding of recursive data structures and generator functions in TypeScript. The recursive nature of the problem suggests that a recursive function may be an elegant solution. The 'inorder traversal' hints at a depth-first search (DFS) approach where for any given array element, we explore as deep as possible along each branch before backing up, which is a typical recursive behavior.

So, the intuitive approach is to iterate over the array, check if the current element is a number or another array.

- If it is a number, we use 'yield' to return this number and pause the function's execution, allowing the numbers to be enumerated one by one.
- If it is another array, we apply recursion and 'yield*' to delegate to the recursive generator. The use of 'yield*' (yield delegation) is crucial because it allows the yielded values from the recursive call to come through directly to the caller. This maintains the proper sequence without needing to manage intermediate collections or states.

This recursive generator pattern allows us to elegantly handle the varying depths of nested arrays and maintain the inorder traversal logic without complicating the process of maintaining the current state during iteration.

Solution Approach

The solution employs a simple yet effective recursive generator function to perform the inorder traversal of a multi-dimensional array.

First, we outline the function signature:

```
function* inorderTraversal(arr: MultidimensionalArray): Generator<number, void, unknown> {
```

We declare a TypeScript generator function `inorderTraversal` that accepts a parameter `arr`, which is of type `MultidimensionalArray`. This type is an alias for an array that can have elements that are either numbers or, recursively, `MultidimensionalArray`.

Next, we implement the core logic:

```
for (const e of arr) {
  if (Array.isArray(e)) {
    yield* inorderTraversal(e);
  } else {
    yield e;
  }
}
```

- The generator function enters a loop, iterating over each element `e` in the input array `arr`.
- For each element, we check if `e` is an array using the `Array.isArray(e)` method. If this is true, it indicates that we have encountered a nested array, so we must traverse it as well, which is done recursively.
- `yield*` is used to delegate to another generator function—this allows the values from the recursive call to be yielded back to the caller of the original generator.

By using `yield` and `yield*`, the elements are yielded one by one, maintaining the expected order of an inorder traversal.

This recursive pattern allows us to succinctly handle the structure irrespective of the depth or complexity of the nesting. It elegantly scales to any level of nested arrays, effectively flattening the structure in an iterative fashion in the order the arrays are laid out. There is no need for additional data structures or managing a stack explicitly as the call stack of the recursive function calls handles this.

Solution Walkthrough

To illustrate the solution approach, let's use a small example of a nested array structure and walk through how the generator function `inorderTraversal` operates on it.

Suppose we have the following multi-dimensional array:

```
const nestedArray = [1, [2, [3, 4], 5], 6, [7, 8]];
```

The `inorderTraversal` function will traverse this nested structure and yield each number sequentially. Here's how it works step-by-step:

1. The function begins the first iteration of the top-level array.
2. It encounters the number `1` and yields it.
3. The next element is `[2, [3, 4], 5]`, which is an array. The function calls itself recursively with this array.
4. Inside this recursive call:
   - It yields `2`.
   - Encounters another nested array `[3, 4]` and recurses again.
   - In the next level of recursion, it yields `3` and `4` sequentially.
   - Having finished with the nested array `[3, 4]`, the function backtracks to the previous level and yields `5`.
5. The function continues with the top-level array and yields `6`.
6. Lastly, it encounters another array `[7, 8]`:
   - It yields `7`.
   - Then it yields `8`.

The final order of yielded numbers reflects an inorder traversal of the multi-dimensional array:

```
1, 2, 3, 4, 5, 6, 7, 8
```

By utilizing recursion, the function correctly and efficiently handles nested structures of any depth without the need for explicitly managing the state or maintaining a stack. The `yield` and `yield*` keywords allow for a clear and concise definition of how values are passed back during the traversal.

Python Solution

```
# Definition for a multidimensional list where each element can either be an integer
# or another multidimensional list of similar structure.
MultidimensionalList = List["MultidimensionalList"] | int

# Generator function to traverse a multidimensional list in order.
# This function yields each integer element encountered during the traversal.
def inorder_traversal(arr: MultidimensionalList):
    # Iterate over each element of the list.
    for element in arr:
        if isinstance(element, list):
            # If the current element is a list, recursively traverse it.
            yield from inorder_traversal(element)
        else:
            # If the current element is an integer, yield it.
            yield element

# Usage example:
# Create a generator instance with a multidimensional list.
generator = inorder_traversal([1, [2, 3]])
# Retrieve numbers from the generator in order.
# next(generator)  # Returns 1
# next(generator)  # Returns 2
# next(generator)  # Returns 3
```

Java Solution

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

// A nested list that can contain either integers or another nested list of integers.
class NestedInteger {
    private Integer value;
    private List<NestedInteger> list;

    public NestedInteger(Integer value) {
        this.value = value;
    }

    public NestedInteger(List<NestedInteger> list) {
        this.list = list;
    }

    public boolean isInteger() {
        return value != null;
    }

    public Integer getInteger() {
        return value;
    }

    public List<NestedInteger> getList() {
        return list;
    }
}

// Iterator to traverse a nested list of integers in order, yielding each number.
class InorderTraversal implements Iterator<Integer> {
    private List<NestedInteger> flattenedList;
    private int currentPosition;

    public InorderTraversal(List<NestedInteger> nestedList) {
        flattenedList = new ArrayList<>();
        flatten(nestedList);
        currentPosition = 0;
    }

    private void flatten(List<NestedInteger> nestedList) {
        for (NestedInteger ni : nestedList) {
            // If the current element is an integer, add it to the flattened list.
            if (ni.isInteger()) {
                flattenedList.add(ni.getInteger());
            } else {
                // If the current element is a nested list, recursively traverse it.
                flatten(ni.getList());
            }
        }
    }

    @Override
    public boolean hasNext() {
        // Return true if there are more elements to iterate over.
        return currentPosition < flattenedList.size();
    }

    @Override
    public Integer next() {
        // Return the next integer in the traversal, if one exists.
        return flattenedList.get(currentPosition++);
    }
}

// Usage example:
/*
NestedInteger nestedList1 = new NestedInteger(1);
NestedInteger nestedList2 = new NestedInteger(Arrays.asList(new NestedInteger(2), new NestedInteger(3)));
List<NestedInteger> nestedList = Arrays.asList(nestedList1, nestedList2);

InorderTraversal traversal = new InorderTraversal(nestedList);
while (traversal.hasNext()) {
    System.out.println(traversal.next()); // Prints 1, then 2, then 3 in order.
}
*/
```

C++ Solution

```
#include <vector>
#include <cstdlib>
#include <iterator>
#include <iostream>

// Type definition for a nested array where each element can either be an integer
// or another nested array with a similar structure.
using MultidimensionalArray = std::vector<std::variant<int, std::vector<int>>>;

// This is a generator function declared with the using syntax.
// It's not supported as a language construct in C++, so we need to simulate it.
// For simplicity, we'll use a recursive lambda function within another function.
std::vector<int> inorderTraversal(MultidimensionalArray& arr) {

    // A lambda function that traverses the array, which captures a local vector by reference.
    // where it would push back elements. In actual generator function, this would directly yield elements.
    std::vector<int> result;
    std::function<void(const MultidimensionalArray&)> traverse = [&](const MultidimensionalArray& sub_arr) {
        for (const auto& element : sub_arr) {
            if (std::holds_alternative<int>(element)) {
                // If the current element is a number, add it to the result.
                result.push_back(std::get<int>(element));
            } else {
                // If the current element is an array, recursively traverse it.
                traverse(std::get<MultidimensionalArray>(element));
            }
        }
    };

    // Start the traversal from the root array.
    traverse(arr);

    // Return the complete result of the traversal.
    return result;
}

// Usage example:
int main() {
    // Create a nested (multidimensional) array.
    MultidimensionalArray arr = {1, MultidimensionalArray{2, 3}};

    // Retrieves the inorder traversal of the array.
    std::vector<int> traversalResult = inorderTraversal(arr);

    // Print the numbers retrieved from the traversal.
    for (int num : traversalResult) {
        std::cout << num << std::endl;
    }

    return 0;
}
```

Typescript Solution

```
// Type definition for a nested array where each element can either be a number
// or another nested array of the same type.
type MultidimensionalArray = (MultidimensionalArray | number)[];

// Generator function to traverse a multidimensional array in order.
// This function yields each number element encountered during the traversal.
function* inorderTraversal(arr: MultidimensionalArray): Generator<number, void, unknown> {
    // Iterate over each element of the array.
    for (const element of arr) {
        if (Array.isArray(element)) {
            // If the current element is an array, recursively traverse it.
            yield* inorderTraversal(element);
        } else {
            // If the current element is a number, yield it.
            yield element;
        }
    }
}

// Usage example:
// Create a generator instance with a multidimensional array.
// const generator = inorderTraversal([1, [2, 3]]);
// Retrieve numbers from the generator in order.
// generator.next().value; // Returns 1
// generator.next().value; // Returns 2
// generator.next().value; // Returns 3
```

Time and Space Complexity

The given TypeScript code performs an inorder traversal on a multidimensional (nested) array to yield all the numbers contained therein. A generator function is defined to achieve this, which allows the user to retrieve the numbers one by one.

Time Complexity

The time complexity of this function can be evaluated by considering that each element in the multidimensional array is visited exactly once. If there are `N` numbers within the nested arrays, regardless of their specific arrangement, each number will be yielded exactly once by the generator function.

While nested arrays necessitate recursive calls, this does not multiply the number of operations per element; rather, it dictates the depth of those recursion calls. Therefore, the time complexity of the function is `O(N)`, where `N` is the total number of elements.

Space Complexity

The space complexity is determined by the maximum depth of recursion needed to reach the innermost array. In the worst-case scenario — that is, a deeply nested array where each array contains only one sub-array until the deepest level — the space complexity is `O(D)`, where `D` is the depth of the nested arrays. This space is used by the call stack of the recursive function.

In a more general scenario with uneven distribution of elements, the space complexity will still be determined by the depth of the recursion required but would be less than the maximum depth of all the elements. Therefore, we can consider the space complexity to be `O(D)`, with an understanding that `D` indicates the depth of the deepest nesting.

If we consider the generator state which also takes up memory, this could add a constant factor to the space complexity, but does not change the overall `O(D)` space complexity.