2625. Flatten Deeply Nested Array

Medium

Leetcode Link

Problem Description

Given a multi-dimensional array `arr` and a depth `n`, we are tasked to create a flattened version of `arr`. A multi-dimensional array contains integers or more multi-dimensional arrays, just like folders that can contain files or more folders. Our goal is to simplify this structure by converting it into a one-dimensional array. The flattening should only occur up to the depth `n`. So, if the depth of an element is less than `n`, we will not flatten that part of the array any further. The depth of the elements in the first array is considered to be `0`.

Intuition

The problem begs for a recursive approach as we are dealing with a structure that is defined in terms of itself—arrays within arrays. The recursion strategy should consider two things each time it is called: the array we are currently working on, and how deep we are, indicated by `n`.

Here's our basic strategy:

- If `n` is `0` or less, we don't do any flattening and return the array as it is. This serves as our base case for the recursion because if we've reached the depth limit, no further flattening should be done.
- We create an empty array to store the flattened elements, `ans`.
- We then iterate over the elements of `arr`. For each element:
  - If it's an array itself, we call the `flat` function recursively on this array, decreasing `n` by `1`. This step gradually works through nested arrays and peels away layers as determined by the depth `n`.
  - If it's not an array (i.e., an integer), we add it directly to `ans`.
- Finally, we return `ans`, which holds the flattened structure up to the `n`-th depth level.

We avoid using the built-in `Array.flat` method, as the problem specification requires us to manually implement the flattening logic.

Solution Approach

The solution uses a recursive function to tackle the problem of flattening a multi-dimensional array to a specific depth (`n`). The algorithm is as follows:

1. **Base Case**:

- Check if `n` is less than or equal to `0`. If this is the case, return the array as-is. This is because we've either reached the desired depth of flattening or don't want to flatten the array at all. The base case prevents further recursion.

2. **Initialization**:

- Initialize an empty array named `ans`. This will hold the partially flattened elements as we process the input array.

3. **Iteration**:

- Iterate through each element `x` of the array `arr` using a `for` loop.

4. **Recursion** and **Concatenation**:

- For each element `x`, check if it's an array using `Array.isArray(x)`.
- If `x` is an array, we need to dive deeper into its content:
  - Recursively call `flat` on element `x`, passing along the decremented depth `n - 1`. This recursive call will return `x` flattened up to `n - 1` layers.
  - Use the spread operator (`...`) to concatenate the elements returned by the recursive call to the `ans` array. This flattens `x` one layer and merges its contents into `ans`.
- If `x` is not an array, simply push `x` onto `ans`.

5. **Return Value**:

- After iterating through all elements, return the `ans` array. The `ans` array represents our multi-dimensional array flattened up to depth `n`.

The solution uses the concept of recursion, coupled with array operations such as iteration and concatenation. The spread operator is particularly useful to merge arrays without introducing additional nesting. This code elegantly avoids the complications of iterative approaches that would require stacks or complex loop control and instead relies on the execution stack from the recursive calls to manage the progression through different array levels.

Example Walkthrough

Let's go through an example to illustrate the solution approach. Assume we are given the following multi-dimensional array `arr` and depth `n`:

```
arr = [[1, 2], [3, [4, 5]], 6]
n = 1
```

We want to flatten `arr` but only up to depth `1`. Here's how the solution would work:

1. **Base Case**: Our function would first check if `n <= 0`. In this case, `n` is `1`, so we proceed with flattening.

2. **Initialization**: We initialize an empty array `ans`. This will hold the partially flattened elements.

3. **Iteration**:
   - We begin by iterating through `arr`. The first element is `[1, 2]`, which is an array.
4. **Recursion** and **Concatenation**:
   - Since the first element `[1, 2]` is an array, we recursively call `flat([1, 2], n - 1)`, which decrements `n` by `1`.
   - The recursive call will return the array `[1, 2]` because `n` now becomes `0`, which hits our base case. This array gets concatenated to `ans`, resulting in `ans = [1, 2]`.
   - The next element in `arr` is `[3, [4, 5]]`, which is another array.
   - We recursively call `flat([3, [4, 5]], n - 1)`.
   - This time, since the first element, `3`, is not an array, it gets added to a new intermediate array. However, the second element `[4, 5]` is an array. Because our depth `n` at this point is `0` (since we decremented before), we add `[4, 5]` as it is without flattening it further. The intermediate result for this recursive call is `[3, [4, 5]]`.
   - This intermediate result is then concatenated to `ans`, which now becomes `ans = [1, 2, 3, [4, 5]]`.
   - The last element in `arr` is `6`. It is not an array, so we directly add `6` to `ans`, resulting in `ans = [1, 2, 3, [4, 5], 6]`.
5. **Return Value**: We have finished iterating through `arr`, and we return `ans`. The final result is a flattened array up to depth `1`:

```
[1, 2, 3, [4, 5], 6]
```

As you can see, the algorithm carefully flattens each layer of the array only up to the specified depth `n` and retains the nested structure beyond that depth. The use of recursion allows us to naturally handle arrays of varying depths without the need for complex loop control or extra data structures.

Python Solution

```python
# Import the typing module for type definitions
from typing import Union, List

# Define a MultiDimensionalArray where an element can be either an integer or another MultiDimensionalArray
MultiDimensionalArray = List[Union[int, 'MultiDimensionalArray']]

def flat(array: MultiDimensionalArray, depth: int) -> MultiDimensionalArray:
    """
    Flatten a multidimensional array 'array' up to a specific 'depth'.
    If the depth is less than or equal to 0, the array is returned as is.

    Parameters:
    array (MultiDimensionalArray): The multi-dimensional array to flatten.
    depth (int): The depth indicating how many levels of nesting to flatten.

    Returns:
    MultiDimensionalArray: The flattened array up to the specified depth.
    """

    # Base case: if the depth is less than or equal to 0, return the array as is.
    if depth <= 0:
        return array

    # Initialize an empty list to store the flattened result
    flattened_array = []

    # Iterate over each element in the input array
    for element in array:
        # Check if the element is a list (array) itself
        if isinstance(element, list):
            # If it is a list, recursively flatten it with a decremented depth, and extend the result to the flattened array
            flattened_array.extend(flat(element, depth - 1))
        else:
            # If the element is an integer, append it to the flattened array
            flattened_array.append(element)

    # Return the result
    return flattened_array
```

Java Solution

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

// Define the type for a MultiDimensionalArray where each element can either be an Integer or another MultiDimensionalArray.
class MultiDimensionalArray extends ArrayList<Object> {}

public class FlattenArray {

    // Define the 'flat' function that takes a MultiDimensionalArray 'array' and
    // a depth 'depth' indicating how many levels of nesting to flatten.
    public static MultiDimensionalArray flat(MultiDimensionalArray array, int depth) {
        // If the specified depth is less than or equal to 0, return the array as is.
        if (depth <= 0) {
            return array;
        }

        // Initialize an empty array to store the flattened result.
        MultiDimensionalArray flattenedArray = new MultiDimensionalArray();

        // Iterate over each element in the input array.
        for (Object element : array) {
            // Check if the element is an array itself.
            if (element instanceof MultiDimensionalArray) {
                // If it is, recursively flatten the array element with reduced depth and append the result.
                flattenedArray.addAll(flat((MultiDimensionalArray) element, depth - 1));
            } else {
                // If it's not an array, append the element as is to the flattened result.
                flattenedArray.add(element);
            }
        }

        // Return the flattened array.
        return flattenedArray;
    }

    // Main method to test the 'flat' function.
    public static void main(String[] args) {
        // Example usage:
        MultiDimensionalArray nestedArray = new MultiDimensionalArray();
        nestedArray.add(1);
        nestedArray.add(new MultiDimensionalArray() {{
            add(2);
            add(new MultiDimensionalArray() {{
                add(3);
                add(4);
            }});
        }});
        nestedArray.add(5);

        // Specify the depth to which the array should be flattened.
        int depth = 1;

        // Flatten the array and print the result.
        MultiDimensionalArray result = flat(nestedArray, depth);
        System.out.println(result);
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <variant>
#include <type_traits>

// Define the variant type for an Element which can either be an integer or a pointer to a MultiDimensionalArray.
using Element = std::variant<int, std::vector<int>*>;

// Define the type for a MultiDimensionalArray where each element can either be an integer or another MultiDimensionalArray pointer.
using MultiDimensionalArray = std::vector<Element>;

// Define the 'Flat' function that takes a MultiDimensionalArray reference 'array'
// and an integer 'depth' indicating how many levels of nesting to flatten.
MultiDimensionalArray Flat(const MultiDimensionalArray& array, int depth) {
    // If the specified depth is less than or equal to 0, return the array as is.
    if (depth <= 0) {
        return array;
    }

    // Initialize an empty array to store the flattened result.
    MultiDimensionalArray flattenedArray;

    // Iterate over each element in the input array.
    for (const Element& element : array) {
        // Check if the element is an integer or another MultiDimensionalArray.
        if (std::holds_alternative<int>(element)) {
            // If it's an integer, append the element as is to the flattened array.
            flattenedArray.push_back(element);
        } else {
            // If it's an array, recursively flatten the nested array element with reduced depth.
            const MultiDimensionalArray& nestedArray = *std::get<std::vector<int>*>(element);
            MultiDimensionalArray flattenedNestedArray = Flat(nestedArray, depth - 1);
            // Append the result to the flattened array.
            flattenedArray.insert(flattenedArray.end(), flattenedNestedArray.begin(), flattenedNestedArray.end());
        }
    }

    // Return the flattened result.
    return flattenedArray;
}
```

Typescript Solution

```typescript
// Define the type for a MultiDimensionalArray where each element can either be a number or another MultiDimensionalArray.
type MultiDimensionalArray = (number | MultiDimensionalArray)[];

// Define the 'flat' function that takes a MultiDimensionalArray 'array' and
// a depth 'depth' indicating how many levels of nesting to flatten.
var flat = function (array: MultiDimensionalArray, depth: number): MultiDimensionalArray {
    // If the specified depth is less than or equal to 0, return the array as is.
    if (depth <= 0) {
        return array;
    }

    // Initialize an empty array to store the flattened result.
    const flattenedArray: MultiDimensionalArray = [];

    // Iterate over each element in the input array.
    for (const element of array) {
        // Check if the element is an array itself.
        if (Array.isArray(element)) {
            // If it is, recursively flatten the array element with reduced depth and append the result.
            flattenedArray.push(...flat(element, depth - 1));
        } else {
            // If it's not an array, append the element as is to the flattened result.
            flattenedArray.push(element);
        }
    }

    // Return the flattened array.
    return flattenedArray;
};
```

Time and Space Complexity

Time Complexity

The time complexity of the `flat` function depends on the size and depth of the input array and the value of `n`. In essence:

- Every element in the original array `arr` will be visited at least once.
- If an element in the array is itself an array, the function will recursively visit each element of that subarray, decrementing `n` by 1 each time.
- This process is repeated until `n` is reduced to 0 or there are no more nested arrays to flatten at the current level of depth.

To represent this formally, let's assume that the size of the array is `m` and the maximum depth of nested arrays is `d`. The function could theoretically traverse every element at every depth up to `n` times (where $n \le d$), leading to a worst-case time complexity of `O(m * d)` when `n` >= `d`. However, the function only flattens `n` levels deep, so the time complexity is strictly bounded by `n` as well, thus it's `O(m * min(d, n))`.

Space Complexity

- There is a space cost incurred every time the `flat` function is recursively called, which contributes to the space complexity. Each recursive call creates a new `ans` array. Given that the maximum depth of recursion is `n`, the space complexity due to recursive calls is `O(n)`.
- Additionally, we also need to consider the space required to store the `ans` array. In the worst case, this could be as large as the total number of elements in the flattened array, which depends on how many elements are at each level of depth and can vary widely.

The total space complexity considering both factors is `O(n + m')`, where `m'` is the total number of individual number elements encountered during the flattening operation up to `n` levels deep.