553. Optimal Division

Medium Array Math Dynamic Programming

Problem Description

The task involves an integer array `nums`, where each adjacent pair of integers performs a floating-point division in the given order. For instance, if `nums` is `[2, 3, 4]`, it represents the expression `2/3/4`. The challenge is to add parentheses to the expression in a way that maximizes the result of the expression. Parentheses can change the order of operations; hence changing the expression's value. The goal is to find where to place these parentheses to get the maximum value expression and return this expression as a string. It's important to note that the final expression must not include unnecessary parentheses.

Intuition

When dividing a set of numbers, placing the larger numbers in the numerator and the smaller ones in the denominator maximizes the result. With division, the key to maximizing the outcome is to divide by the largest possible number. Based on this logic, we want to minimize the denominator as much as possible to maximize the result of the entire expression.

For three or more numbers, the maximum result is achieved when we divide the first number by the result of dividing all the remaining numbers. Therefore, for `nums = [a, b, c, ..., n]`, the optimal division is `a/(b/c/.../n)`. This ensures that `a` is divided by the smallest possible value obtained from the division of the rest of the numbers. In this case, we only need parentheses around all the numbers starting from the second number in the array.

For `nums` with only one element, there is no division to perform, so we return the single number. When `nums` has exactly two elements, say `[a, b]`, the expression is simply `a/b`, as there's no other way to place parentheses that can alter the result. The solution naturally emerges from these observations:

1. If `nums` contains only one element, that is the expression.
2. If `nums` contains two elements, the expression is `"a/b"`.
3. For three or more elements, we use parentheses to ensure that the first element is divided by the smallest possible value obtained from the division of the rest of the numbers: `"a/(b/c/.../n)"`.

The provided Python solution implements these insights through conditional logic and string formatting.

Solution Approach

The provided Python solution implements a direct approach without the need for any complex algorithms or data structures. Here's how the code accomplishes the task:

1. The solution first checks the length of `nums`:
   - If the array has only one element (`n == 1`), it returns that element as a string.
   - If the array has two elements (`n == 2`), it returns the division of the first element by the second as a string formatted `"a/b"`.
2. For arrays with three or more elements (`n >= 3`), the solution constructs an expression that places the first number in the numerator and the rest in the denominator within a pair of parentheses. This is based on the intuition that to maximize the value of the division, you want to divide the first number by the result of the subsequence divisions, thus minimizing the denominator.
3. The code uses Python's string formatting and string `join` method to construct the final expression. The part `"/".join(map(str, nums[1:]))` joins all elements of `nums` starting from the second element with a division sign between them. It then formats the entire string by placing the first element of `nums` outside and the joined string inside the parentheses.

No additional data structures are required since the final expression is constructed directly from the input list. The solution takes advantage of Python's string manipulation capabilities to format the output as per the problem's requirements.

The simplicity of the solution comes from the mathematical insight that the highest value for the entire expression is obtained when the first element of `nums` is divided by the smallest possible value from the division of the subsequent elements. This is efficiently implemented in the given Python function `optimalDivision`.

Here's the implementation logic step by step in the solution:

```
class Solution:
    def optimalDivision(self, nums: List[int]) -> str:
        n = len(nums)
        if n == 1:
            return str(nums[0])                  # Case for single element, return as string
        if n == 2:
            return f'{nums[0]}/{nums[1]}'        # Case for two elements, return "a/b"
        # Case for three or more elements, return "a/(b/c/.../n)"
        return f'{nums[0]}/({"/".join(map(str, nums[1:]))})'
```

This concise implementation elegantly solves the problem by directly translating the mathematical insight of division into a formatted string, which is exactly what the problem statement is asking for.

Example Walkthrough

Let's go through a small example using the solution approach to understand how the algorithm maximizes the expression by correctly placing the parentheses.

Suppose we have the array `nums = [8, 2, 3, 4]`, which represents the expression `8/2/3/4`. The task is to add parentheses to maximize this expression's value.

Following the solution approach:

1. We first check the length of `nums`. Since `n = 4`, we have more than two elements.
2. For an array with more than two elements, the goal is to maximize the first number's division by the smallest possible result from subsequence divisions. To achieve this, we should divide the first element by the result of dividing all the subsequent numbers. This translates to the expression: `8/(2/3/4)`.
3. To create this expression in Python, the code joins all elements of `nums`, starting from the second element, with a division sign, resulting in the string `"2/3/4"`.
4. Then, the code formats the entire expression by placing the first element of `nums` (`8`) outside and the joined string inside parentheses, resulting in the final expression `"8/(2/3/4)"`.

Using this approach, we have successfully maximized the value of the original expression. In the absence of parentheses, the expression would calculate as follows: `8/2/3/4 = 4/3/4 = 1.33`. However, by placing parentheses as `8/(2/3/4)`, we get: `8/(0.166...) = 48`, which is indeed the maximum value possible for this expression.

The Python code effectively and efficiently constructs this optimal expression using string formatting without the need for additional computation or complex data structures.

Solution Implementation

Python

```
from typing import List

class Solution:
    def optimalDivision(self, nums: List[int]) -> str:
        # Calculate the length of the input list of numbers
        size = len(nums)

        # If there is only one number, return it as a string
        if size == 1:
            return str(nums[0])

        # If there are two numbers, return them as a division
        if size == 2:
            return f'{nums[0]}/{nums[1]}'

        # For more than two numbers, the optimal division is to divide the first number
        # by the result of the division of all remaining numbers to achieve the largest result.
        # This is done by grouping all but the first number inside parentheses.
        return f'{nums[0]}/(' + "/".join(map(str, nums[1:])) + ')'
```

Java

```
class Solution {
    // Method to find the optimal division of integers as a string
    public String optimalDivision(int[] nums) {
        // Get the number of elements in the array
        int arrayLength = nums.length;

        // If there is only one number, return it as there's nothing to divide
        if (arrayLength == 1) {
            return String.valueOf(nums[0]);
        }

        // If there are two numbers, return their division
        if (arrayLength == 2) {
            return nums[0] + "/" + nums[1];
        }

        // If there are more than two numbers, we use parentheses to maximize the result
        StringBuilder result = new StringBuilder();

        // Start the string with the first number and an opening parenthesis
        result.append(nums[0]).append("/(");

        // Loop through the rest of the array, except for the last element
        for (int i = 1; i < arrayLength - 1; ++i) {
            // Add each number followed by a division sign
            result.append(nums[i]).append("/");
        }

        // Add the last element of the array and the closing parenthesis
        result.append(nums[arrayLength - 1]).append(")");

        // Return the constructed string
        return result.toString();
    }
}
```

C++

```
class Solution {
public:
    // This function finds the optimal division of array numbers as a string expression
    string optimalDivision(vector<int>& nums) {
        int n = nums.size();

        // If there is only one number, return it as a string
        if (n == 1) {
            return to_string(nums[0]);
        }

        // If there are two numbers, return their division
        if (n == 2) {
            return to_string(nums[0]) + "/" + to_string(nums[1]);
        }

        // For more than two numbers, enclose all but the first number in parentheses
        // This is to ensure the division is performed correctly for optimal result
        string answer = to_string(nums[0]) + "/(";

        // Append all the remaining numbers separated by a division operator
        for (int i = 1; i < n - 1; i++) {
            answer.append(to_string(nums[i]) + "/");
        }

        // Add the last number and close the parentheses
        answer.append(to_string(nums[n - 1]) + ")");

        return answer;
    }
};
```

TypeScript

```
/**
 * Returns a string representation of the division of numbers that maximizes the result.
 * If there are more than two numbers, brackets are added to divide the first number by
 * the result of the division of all subsequent numbers.
 * @param {number[]} numbers - An array of numbers to divide.
 * @returns {string} - The string representation of the optimal division.
 */
function optimalDivision(numbers: number[]): string {
    // Get the number of elements in the array
    const count = numbers.length;

    // Combine the numbers into a string separated by slashes
    const divisionString = numbers.join('/');

    // If there are more than two numbers, add brackets after the first division
    if (count > 2) {
        // Find the index of the first slash to insert the opening bracket
        const firstSlashIndex = divisionString.indexOf('/') + 1;
        // Return the string with brackets inserted
        return `${divisionString.slice(0, firstSlashIndex)}(${divisionString.slice(firstSlashIndex)})`;
    }

    // If there are two or fewer numbers no brackets are needed
    return divisionString;
}
```

```
from typing import List

class Solution:
    def optimalDivision(self, nums: List[int]) -> str:
        # Calculate the length of the input list of numbers
        size = len(nums)

        # If there is only one number, return it as a string
        if size == 1:
            return str(nums[0])

        # If there are two numbers, return them as a division
        if size == 2:
            return f'{nums[0]}/{nums[1]}'

        # For more than two numbers, the optimal division is to divide the first number
        # by the result of the division of all remaining numbers to achieve the largest result.
        # This is done by grouping all but the first number inside parentheses.
        return f'{nums[0]}/(' + "/".join(map(str, nums[1:])) + ')'
```

Time and Space Complexity

Time Complexity

The time complexity of the code primarily depends on the length of the `nums` array. The `map` function iterates over `nums[1:]`, which has a complexity of `O(n-1)` where `n` is the length of the array. The `join` operation on a list of strings has a complexity of `O(m)`, where `m` is the total length of the strings being concatenated. Since in this case we're dealing with string representations of the numbers, the length of each integer after conversion to string can vary, but in the worst case, it will be proportional to the logarithm of the number. However, for the purpose of time complexity analysis, we can consider `m` to be linearly dependent on `n` because the number of divisions does not change the overall complexity class. Hence, the `join` operation is also `O(n)`. The overall time complexity is `O(n)`.

Space Complexity

The space complexity includes the space required for the output and the temporary lists created by `map` and `join`. The size of the output string is `O(n)` since it includes all the elements of `nums` plus additional characters. The temporary list created by `map` is also `O(n)`. However, these are not additional spaces in terms of space complexity analysis since they are required for the output. Thus, the space complexity is `O(n)`.