2221. Find Triangular Sum of an Array

Medium Array Math Combinatorics Simulation

Leetcode Link

Problem Description

The problem presents us with an integer array `nums` with each element being a digit from 0 to 9. We need to compute what is called the "triangular sum" of this array.

To find the triangular sum, we repeatedly perform the following process:

1. Check if the array `nums` has only one element. If yes, the process ends, and this element is our triangular sum.
2. If the array has more than one element, we create a new array `newNums` that has one less element than `nums`.
3. We then populate `newNums` such that each element at index `i` is equal to `(nums[i] + nums[i+1]) % 10`. This is the sum of consecutive elements in the original array modulo 10.
4. We replace `nums` with `newNums` and repeat the process from step 1.

Our goal is to keep repeating this process until there is only one element left in `nums`, which will be our answer.

Intuition

The solution is based on iterative reduction. Notice that in each iteration, the array size is reduced by 1. This process continues until only one element remains. Our task is to simulate this process programmatically.

The approach avoids creating multiple arrays for each iteration by continually updating the original array, which is both space-efficient and direct. It leverages in-place updates for `nums`, each time updating the element at index `i` based on the element at `j` and `j + 1`.

We initiate this process from the first element (index `0`) to the second last element (`n - 2`) of the current iteration's array size, and afterward, we reduce our array size (`n`) by 1. This process is repeated until we reach the array size of 1, and this final single element is the triangular sum of the original array.

Solution Approach

The reference solution code provided demonstrates the implementation of the intuitive approach we discussed earlier. It utilizes a nested loop to successively reduce the array and calculate the new values.

Here's a step-by-step breakdown of the algorithm applied in the solution code:

1. Determine the length of the `nums` array and store it in the variable `n`.
2. Start an outer loop to iteratively reduce the size of the `nums` array. This loop runs from the current size `n` down to `1`.
3. Inside the outer loop, there's an inner loop which is where the actual update takes place. This loop runs from `0` to `i - 1`, where `i` is the current size of the `nums` array for that iteration.
4. Within the inner loop, calculate the new value for `nums[j]` as `(nums[j] + nums[j + 1]) % 10`. This formula is applied to every pair of adjacent elements, effectively shifting the entire `nums` array by one position to the left.
5. When the outer loop finishes (which is when the array size `n` reaches 1), we have our triangular sum stored in `nums[0]`, and we return this value as the triangular sum of the array.

The data structure used here is just the initial integer array `nums`, and we are manipulating it in place to maintain space efficiency. No auxiliary data structures are used.

The pattern followed is iterative reduction combined with in-place updates, which is a common way to deal with problems where the size of the data structure changes in each iteration based on specific calculation rules.

Here is the core part of the solution code that aligns with the description above:

```
n = len(nums)
for i in range(n, 0, -1):
    for j in range(i - 1):
        nums[j] = (nums[j] + nums[j + 1]) % 10
return nums[0]
```

As seen, the outer loop runs until the array is reduced to a single element, and the inner loop makes the required modifications to the `nums` array based on the transition rule `(nums[i] + nums[i+1]) % 10`.

This approach is both clean and efficient, allowing for easy understanding and avoiding any unnecessary space complexity.

Example Walkthrough

Let's take a small example to illustrate the solution approach. Suppose `nums` is `[1, 2, 3, 4, 5]`. We need to apply the solution approach to this input to find the triangular sum.

1. `n = len(nums)` will set `n` to be `5` since there are five elements in the input array.
2. We then enter the outer loop, which will continue until `n` is reduced to `1`.
3. The first iteration of the outer loop will not change `n`, but will initiate the inner loop which runs `n-1` times (4 times in this case).
4. In the inner loop for this iteration, we update `nums` as follows:
   - `nums[0]` becomes `(1 + 2) % 10 = 3`
   - `nums[1]` becomes `(2 + 3) % 10 = 5`
   - `nums[2]` becomes `(3 + 4) % 10 = 7`
   - `nums[3]` becomes `(4 + 5) % 10 = 9` Now, `nums` looks like `[3, 5, 7, 9]`.
5. For the next iteration of the outer loop, we decrement `n` to `4` and run the inner loop again to update `nums`:
   - `nums[0]` becomes `(3 + 5) % 10 = 8`
   - `nums[1]` becomes `(5 + 7) % 10 = 2`
   - `nums[2]` becomes `(7 + 9) % 10 = 6` Now, `nums` looks like `[8, 2, 6]`.
6. With `n` now at `3`, the outer loop iterates and the inner loop continues updates:
   - `nums[0]` becomes `(8 + 2) % 10 = 0`
   - `nums[1]` becomes `(2 + 6) % 10 = 8` Now, `nums` looks like `[0, 8]`.
7. Lastly, with `n` at `2`, we run the inner loop one more time:
   - `nums[0]` becomes `(0 + 8) % 10 = 8` Now `nums` is simply `[8]`.
8. When we exit the outer loop, we return `nums[0]`, which is `8`. This is the triangular sum of the array `[1, 2, 3, 4, 5]`.

Following the problem's strategy, we successfully computed the triangular sum by iteratively reducing the array and updating it in place without any extra space. The final answer is `8`.

Python Solution

```
# Import the typing module to define the type of input.
from typing import List

class Solution:
    def triangular_sum(self, nums: List[int]) -> int:
        """
        Calculate the triangular sum of a list of integers.
        The triangular sum of a list of integers is obtained by the following process:
        - Replace each element by the sum of itself and the next element, modulo 10.
        - Repeat the process until only one number remains.

        Args:
        nums (List[int]): A list of integers between 0 and 9 inclusive.

        Returns:
        int: The final integer (between 0 to 9) after the process is done.
        """
        # Get the initial length of nums
        n = len(nums)

        # Loop from the length of nums down to 1
        for i in range(n, 0, -1):
            # Calculate the new values for the triangular sum step
            for j in range(i - 1):
                # Update each element with the sum of itself and the next element, mod 10
                nums[j] = (nums[j] + nums[j + 1]) % 10

        # The first element of the list is the answer after the process
        return nums[0]

# Note: The method name has been changed to 'triangular_sum' to match the Python naming convention.
# All variable names are already properly named, and no method names have been altered.
```

Java Solution

```
class Solution {
    public int triangularSum(int[] nums) {
        // Find the length of the input array.
        int n = nums.length;

        // Outer loop to reduce the array size by 1 in each iteration.
        // Note that it should be `i > 0` rather than `i >= 0`
        // since we want to stop when we reach the last element.
        for (int i = n; i > 0; --i) {
            // Inner loop to iterate through the elements up to the new size.
            for (int j = 0; j < i - 1; ++j) {
                // Update the current element by adding it to the next element.
                // The sum is modulo 10 as per the problem's requirement.
                nums[j] = (nums[j] + nums[j + 1]) % 10;
            }
        }

        // After reducing the array to a single element, return that element.
        return nums[0];
    }
}
```

C++ Solution

```
#include <vector> // Include the header for using vectors

class Solution {
public:
    int triangularSum(std::vector<int>& nums) {
        // Get the size of the input vector `nums`
        int size = nums.size();

        // Start from the last row and move upwards, considering each row
        for (int row = size; row > 0; --row) {
            // Iterate through each element up to the second-to-last of the current row
            for (int col = 0; col < row - 1; ++col) {
                // Update the current element by adding it with the next element,
                // and apply modulo 10 to the result
                nums[col] = (nums[col] + nums[col + 1]) % 10;
            }
        }

        // After processing all rows, the first element of the vector contains the result
        return nums[0];
    }
};
```

Typescript Solution

```
// Import the array utility module
import { array } from "util";

// Function that calculates the triangular sum of an array of numbers
function triangularSum(nums: number[]): number {
    // Get the length of the input array `nums`
    let size: number = nums.length;

    // Start from the last row and move upwards, considering each row
    for (let row = size; row > 0; --row) {
        // Iterate through each element up to the second-to-last of the current row
        for (let col = 0; col < row - 1; ++col) {
            // Update the current element by adding it with the next element,
            // and apply modulo 10 to the result
            nums[col] = (nums[col] + nums[col + 1]) % 10;
        }
    }

    // After processing all rows, the first element of the array contains the triangular sum
    return nums[0];
}

// Sample usage of the function
const sampleArray: number[] = [1, 2, 3, 4, 5];
const result: number = triangularSum(sampleArray);
console.log(`The triangular sum is: ${result}`);
```

Time and Space Complexity

The time complexity of the provided code can be analyzed as follows:

We have a nested loop where the outer loop runs `n` times (where `n` is the length of `nums`) and the inner loop performs `i - 1` iterations, where `i` decreases from `n` to 1. This creates a triangular number of operations (hence the name "triangularSum"). The total number of operations can be calculated by the sum of the first `n` integers which is `(n * (n + 1)) / 2`. Consequently, removing the non-dominant terms and constants, we can describe the time complexity as `O(n^2)`.

The space complexity of the code can be determined as follows:

The solution modifies the input list `nums` in-place and does not use any additional space that grows with the input size (other than a constant amount of space for the loop indices and the variable `n`). Hence, the space complexity is `O(1)`.