1304. Find N Unique Integers Sum up to Zero

Easy Array Math

Leetcode Link

Problem Description

The problem presents a task where you are required to generate an array with a size equal to the given integer `n`. This array should consist of `n` distinct integers such that their sum equals zero. In other words, if you were to add up all the elements of the array, the resulting sum should be `0`. The condition for the integers to be unique means no number is repeated within the array.

Intuition

The intuition behind the provided solution leverages the mathematical concept that if you have a set of numbers and their negatives, their total sum will be zero. To achieve this for an array of `n` unique integers, you can create a sequence of integers from `1` to `n-1`. These are your positive numbers. The sum of these numbers will, of course, not be zero. So, to balance out the sum to zero, you need a negative number that is equal to the negative sum of all the positive numbers you've included so far. By appending this balancing negative number to the array, the sum of the entire array becomes zero, fulfilling the problem's requirement.

For example, if `n` is `5`, you might start with `[1, 2, 3, 4]`. The sum of these numbers is `10`. To have a sum of zero, you need to add `-10` to the array. Therefore, the resulting array would be `[1, 2, 3, 4, -10]`, which satisfies the problem's conditions by being unique integers that add up to `0`.

This method works effectively for all `n`, except when `n` is `1`, in which case the answer is just `[0]`, because the array must contain a single integer that sums to zero. The provided code does not explicitly handle this case because `[0]` is effectively covered by the current implementation: when `n` is `1`, the `range(1, n)` produces an empty list, and the negative sum of an empty list is `0`.

Solution Approach

The solution shown involves a simple yet clever use of a mathematical approach combined with Python's built-in functions and list data structure. Here is a step-by-step breakdown:

1. **Create a Sequence of Integers**: Using the `range` function, we create a sequence from `1` to `n-1`. This is done using the expression `list(range(1, n))`. The `list` function then takes this range and turns it into a list of integers.

2. **Compute the Sum of Sequence**: Before we append the final integer to balance our array to sum to `0`, we calculate the sum of all the current integers in the array. This is done with the `sum` function like so: `sum(ans)`.

   - **Note**: At this point, `ans` contains numbers from `1` to `n-1`. Since we haven't added the last number to balance the sum to zero yet, we refer to the current list as `ans`.
3. **Find the Balancing Integer**: The last integer needed to make the sum of the array `0` is the negative of the sum calculated in step 2. Therefore, it is found with `-sum(ans)`.

4. **Append the Balancing Integer to Array**: We append this balancing integer to our array `ans` which at this point has `n-1` integers. By appending the negative sum, we now have `n` integers whose sum is `0`.

5. **Return the Array**: The final step is to return the updated array `ans` which now contains `n` unique integers that sum up to `0`.

In terms of data structures, this solution uses a list to hold the sequence of unique integers. The algorithm itself is simple and does not require complex patterns or operations.

It is interesting to note that this approach is efficient and does not require sorting or set operations that could increase the time complexity. The overall time complexity is O(n) because we iterate over a sequence of numbers that increase linearly with `n`, and the `sum` function also operates in O(n). The space complexity is O(n) as well since we are storing `n` integers in the list.

Example Walkthrough

To illustrate the solution approach, let's walk through a small example where `n = 4`.

1. **Create a Sequence of Integers**: We begin by creating a list of integers from `1` to `n-1` using `range(1, n)`. Since `n` is `4`, our range will be from `1` to `3` (because `range` function in Python is up to, but not including, the end number). The result is a list: `[1, 2, 3]`.

2. **Compute the Sum of Sequence**: We calculate the sum of all the integers currently in the list. The `sum` function will give us `1 + 2 + 3 = 6`. So, the current sum of the list is `6`.

3. **Find the Balancing Integer**: Since the sum of our list is `6`, we need an integer that, when added to this sum, will give us `0`. The opposite of `6` is `-6`, so `-6` is our balancing integer.

4. **Append the Balancing Integer to Array**: We now take `-6` and append it to our current list of `[1, 2, 3]`. After appending, our list becomes `[1, 2, 3, -6]`.

5. **Return the Array**: The final step is to return our updated list, which is `[1, 2, 3, -6]`. This fulfills the requirement of the problem, as we have a list of `n` unique integers whose sum is `0`.

With these steps, regardless of the value of `n`, we can confidently generate a list of `n` unique integers that sum to zero, with the only special case being when `n = 1`, which, as mentioned, inherently returns `[0]`. The simplicity and cleverness of the approach is efficient and neatly sidesteps any potentially complex operations.

Python Solution

```python
class Solution:
    def sumZero(self, n: int) -> List[int]:
        # Initialize an empty list to store our unique numbers
        unique_numbers = list(range(1, n))

        # Add the negative of their sum to the list to ensure the final sum is zero
        # This works because the sum of all numbers from 1 to n-1 is (n-1)*n/2, so adding
        # its negative will cancel out the sum, resulting in zero.
        unique_numbers.append(-sum(unique_numbers))

        # Return the list of numbers which will sum to zero
        return unique_numbers

```

Java Solution

```java
class Solution {
    public int[] sumZero(int n) {
        // Initialize an array to hold 'n' elements
        int[] result = new int[n];

        // Assign values from 1 to n-1 to the array elements
        for (int i = 1; i < n; ++i) {
            result[i] = i;
        }

        // Calculate the sum of elements from 1 to n-1
        // To ensure the sum of the entire array is zero,
        // assign the negative of this sum to the first element
        result[0] = -(n * (n - 1) / 2);

        // Return the array with elements that sum to zero
        return result;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <numeric> // Include the numeric header for std::iota function

class Solution {
public:
    // Function to generate a vector of 'n' integers that sum up to zero
    vector<int> sumZero(int n) {
        // Create a vector with 'n' elements to hold the result
        vector<int> result(n);

        // Populate the vector with consecutive integers starting from 1
        std::iota(result.begin(), result.end(), 1); // will fill the vector with 1, 2, ... n-1

        // Calculate the negative of the sum of first (n-1) integers.
        // The formula for the sum of the first (n-1) positive integers is: sum = (n-1)*n/2.
        // This ensures that the sum of the entire array will be zero.
        result[n - 1] = -(n - 1) * n / 2;

        // Return the result vector
        return result;
    }
};

```

Typescript Solution

```typescript
function sumZero(n: number): number[] {
    // Create an array to store the answer, filled with zeros initially
    const answer = new Array<number>(n).fill(0);

    // Populate the array with numbers 1 to n-1
    for (let i = 1; i < n; ++i) {
        answer[i] = i;
    }

    // Calculate the negative value that will balance the sum of the sequence to zero
    // The sum of the first n-1 positive integers is (n-1) * (n)/2, so we need the negative of that
    answer[0] = -((n * (n - 1)) / 2);

    // Return the resulting array
    return answer;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given function is `O(n)`, where `n` is the input to the function. This is because the function has two main operations: generating a list of `n-1` integers and calculating the sum of the generated list which it negates and appends to the list. Generating a list of `n-1` integers takes `O(n)` time, and calculating the sum of the list also takes `O(n)` time. However, since these operations are sequential, the overall time complexity does not compound and remains `O(n)`.

Space Complexity

The space complexity of the function is also `O(n)`. This is because the function creates a list to store `n` integers. The list starts with `n-1` integers and one more integer is appended to it at the end. Thus, the amount of space needed grows linearly with the input `n`, resulting in a space complexity of `O(n)`.