1228. Missing Number In Arithmetic Progression

Easy Array Math Leetcode Link

Problem Description

The problem provides an array `arr` where the values were originally in an arithmetic progression. An arithmetic progression is a sequence of numbers in which the difference between consecutive terms is constant. This constant difference is missing in the current array because one of the values, which was neither the first nor the last, has been removed.

Our task is to find out which value was removed from the array, knowing that the remaining values continue to have a constant difference between them, except for the place where the value was removed.

Intuition

The intuition behind the solution comes from the properties of an arithmetic progression. In a complete arithmetic progression with `n` terms, the average of the first term `a1` and the last term `an` is also the average of all terms in the sequence. Thus, the sum of the entire progression can be calculated using the formula:

```
sum = average * number_of_terms
sum = ((a1 + an) / 2) * n
```

If a term in the middle is missing, the sum of the terms that are present will be less than the expected sum of the full arithmetic progression by exactly the value of the missing term. We can rearrange the formula to find the missing term as follows:

```
missing_term = expected_sum - actual_sum
```

The solution approach uses this fact to find the missing number. It calculates the expected sum by using the formula with `len(arr) + 1` terms (since one term is missing) and then subtracts the actual sum of the array `arr`. What we are left with is the value of the missing term. The code is a direct implementation of this logic.

Solution Approach

The solution uses a direct mathematical approach and does not rely on complex data structures, algorithms, or patterns. It is a straightforward translation of the mathematical insight into Python code.

Here are the steps the code takes to implement the solution:

1. Calculate the expected sum of the arithmetic progression if it were complete. This is done using the formula `((arr[0] + arr[-1]) * (len(arr) + 1)) // 2`. The `arr[0]` represents the first element, and `arr[-1]` represents the last element in the list. Since one item is missing, the original length before removal would have been `len(arr) + 1`.
2. Calculate the actual sum of the available elements in the array using the built-in `sum(arr)` function.
3. The difference between the expected sum of a complete arithmetic progression and the actual sum of the given array represents the value of the missing element. So, `missing_number = expected_sum - actual_sum`.

The Python code for the solution method `missingNumber` implements these steps, and this is why the entire logic of finding the missing number is condensed into a single line:

```
return (arr[0] + arr[-1]) * (len(arr) + 1) // 2 - sum(arr)
```

By executing this line of code, we calculate and return the missing element directly. The use of integer division `//` ensures that the result is an integer, which matches the problem's requirement that elements of the array are integers.

Example Walkthrough

Let's go through a small example to illustrate the solution approach:

Suppose we have an array `arr` representing an arithmetic progression with one value removed:

```
arr = [5, 7, 11, 13]
```

To find the missing number, we would:

1. **Calculate the expected sum of the arithmetic progression if it were complete.**

In this example, the first element `a1` is 5, and the last element `an` is 13. The length of the original array before a number was removed would be `len(arr) + 1 = 5`. Using the formula, the expected sum is:

```
expected_sum = ((a1 + an) / 2) * n = ((5 + 13) / 2) * 5 = (18 / 2) * 5 = 9 * 5 = 45
```

2. **Calculate the actual sum of the available elements in the array.**

Now we use the sum function to find the actual sum:

```
actual_sum = sum(arr) = 5 + 7 + 11 + 13 = 36
```

3. **Calculate the difference between the expected sum and the actual sum to find the missing term.**

Next, we find the missing term:

```
missing_number = expected_sum - actual_sum = 45 - 36 = 9
```

The missing term in the arithmetic progression is 9. We can check this by adding the missing term to the array:

```
arr = [5, 7, 9, 11, 13]
```

Now `arr` is a complete arithmetic progression with a common difference of 2 between each term.

Python Code Implementation

Applying the example into the Python code, we would have:

```python
def missingNumber(arr):
    return (arr[0] + arr[-1]) * (len(arr) + 1) // 2 - sum(arr)

# Given array
arr = [5, 7, 11, 13]

# Call the function to find the missing number
print(missingNumber(arr))  # Output: 9
```

By running this code with our example array `[5, 7, 11, 13]`, we find that the output is `9`, which confirms that the missing number has been correctly identified using the solution approach.

Python Solution

```python
from typing import List

class Solution:
    def missing_number(self, nums: List[int]) -> int:
        # Calculate the expected sum of the arithmetic series using the formula
        # S = n/2 * (first_element + last_element)
        # Here, 'n' is the number of elements the array is supposed to have,
        # which is one more than the current number of elements due to the missing number
        n = len(nums) + 1
        expected_sum = n * (nums[0] + nums[-1]) // 2

        # Subtract the actual sum from the expected sum to find the missing number
        actual_sum = sum(nums)
        missing_number = expected_sum - actual_sum

        return missing_number

# Example usage:
# sol = Solution()
# print(sol.missing_number([3, 0, 1]))  # It should return the missing number in the sequence

```

Java Solution

```java
import java.util.Arrays;

class Solution {
    // Method to find the missing number in the sequence
    public int missingNumber(int[] nums) {
        // Calculate the expected length of the series including the missing number
        int length = nums.length;

        // Compute the expected sum of the series using the arithmetic series formula:
        // Sum = (first number + last number) * number of terms / 2
        // Since the array is missing one number, we consider the length as (length + 1)
        // Here we assume the series starts at 0 and ends with length, hence we add only arr[0]
        int expectedSum = (0 + length) * (length + 1) / 2;

        // Compute the actual sum of the array's elements
        int actualSum = Arrays.stream(nums).sum();

        // The missing number is the difference between the expected and actual sums
        return expectedSum - actualSum;
    }
}

```

C++ Solution

```cpp
#include <vector>       // Include necessary header for vector usage
#include <numeric>      // Include header for std::accumulate function

class Solution {
public:
    // Function to find the missing number in an arithmetic progression
    int missingNumber(std::vector<int>& nums) {
        int size = nums.size(); // Store the size of the array

        // Calculate the sum of the first and last elements in the array
        // and multiply by the count of numbers in the complete sequence (size + 1)
        // then divide by 2 to get the expected sum of the sequence if it were complete
        int expectedSum = (nums[0] + nums[size - 1]) * (size + 1) / 2;

        // Compute the actual sum of the elements in the given array
        int actualSum = std::accumulate(nums.begin(), nums.end(), 0);

        // The difference between expected and actual sum is the missing number
        return expectedSum - actualSum;
    }
};

```

Typescript Solution

```typescript
// Import the 'reduce' method for array summation
import { reduce } from 'lodash';

// Function to find the missing number in an arithmetic progression
function missingNumber(nums: number[]): number {
    let size: number = nums.length; // Store the length of the array

    // Calculate the sum of the first and last elements in the array
    // and multiply by the count of numbers in the complete sequence (size + 1)
    // then divide by 2 to get the expected sum of the sequence if it were complete
    let expectedSum: number = (nums[0] + nums[size - 1]) * (size + 1) / 2;

    // Compute the actual sum of the elements in the given array
    let actualSum: number = reduce(nums, (sum, value) => sum + value, 0);

    // The difference between the expected and actual sum is the missing number
    return expectedSum - actualSum;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is `O(n)`, where `n` is the length of the input array `arr`. This is because the primary operation that depends on the size of the input is the `sum(arr)` function, which iterates through each element of the array once to compute the sum.

Other operations, like computing `arr[0] + arr[-1]` and `len(arr) + 1`, are executed in constant time, `O(1)`, meaning that their execution time does not depend on the size of the input array.

Space Complexity

The space complexity of the code is `O(1)`.

This is because the code uses a fixed amount of space regardless of the input size. The space used for storing the result of `arr[0] + arr[-1]` and the intermediate calculations for `(len(arr) + 1) // 2 - sum(arr)` does not scale with the size of the input array; it remains constant.