



Problem Description

In this problem, you're given an array nums which contains a list of integers. The task is to calculate a running sum of this array. The running sum is a new array where each element at index i is the sum of all the numbers in the nums array up to and including the number at index i.

More formally, the running sum of nums is an array where the value at the i-th index is equal to sum(nums[0]...nums[i]). So for runningSum[i], we add up all the elements from the start of the array to the current element at i.

Your goal is to return the running sum of the given array nums.

Intuition

To solve this problem, we can use a common algorithm called *Prefix Sum*. The idea behind Prefix Sum is simple – as we move through the array element by element, we keep updating the current element to be the sum of itself and all previous elements.

So how do we arrive at this solution? Let's think about the process step by step. If we were doing this by hand, we'd start at the first element - nothing to add here since it's the start. We note this number down. Then, we'd move to the next number, add it to the first, and note this new sum down. Then, for the third number, we'd add it to the new sum we just computed, and so on.

This is exactly what the Prefix Sum algorithm does. We start with the first element, then for each subsequent element at index i, we add the current element nums [i] to the sum we have obtained so far, which is the previous element nums [i-1]. This updated sum becomes the new value for nums [i].

By following this approach, we only need to traverse the array once, making our algorithm efficient and straightforward.

Solution Approach

The solution provided uses a built-in Python function accumulate from the itertools module, which essentially applies the Prefix Sum algorithm automatically for us.

Here's the step by step approach using the Prefix Sum concept:

- 1. Initialize a variable, let's call it prefixSum, to store the sum of elements we've encountered so far. Initially prefixSum is set to 0.
- 2. Traverse the array nums from the first to the last element.
- 3. For each element at index i, update prefixSum by adding the current element nums[i] to it. This will hold the sum of all elements up to the current index.
- 4. Assign the value of prefixSum back to nums[i] to reflect the running sum till the current element.
- 5. Repeat steps 3-4 for the entire length of the array.
- 6. Once the loop ends, we'd have transformed the original nums array into the running sum array.

Now, in the provided solution, the step-by-step process is neatly wrapped up by Python's accumulate() function. This function takes an iterable (in our case, the array nums) and returns an iterator that yields accumulated sums (or other binary functions if specified).

The solution could also be implemented more explicitly, without the use of accumulate(), which would look something like this:

```
def runningSum(self, nums: List[int]) -> List[int]:
    for i in range(1, len(nums)):
        nums[i] += nums[i-1]
    return nums
```

In the above explicit implementation:

1 class Solution:

- We use a for loop to traverse the array starting from the second element (since the first element's running sum is the element itself).
- We then continuously update each element with the sum of itself and all previous elements. This is done by adding the current element nums [i] with the previous element nums [i-1] and storing the result back in nums [i].

After traversing through the array, we return the nums array which now contains the running sum.

Both approaches accomplish the task using the powerful concept of Prefix Sum, but one is more succinct by leveraging Python's built-in functionality.

Let's walk through a small example to make the solution approach clear. Suppose the nums array is [3, 1, 2, 10].

10 which equals 16.

Example Walkthrough

Here are the steps we would take to get the running sum:

1. Start with the first element. Since there are no elements before it, the running sum is just the element itself. So runningSum[0]

- would be 3. 2. Move to the second element, 1. We add this to the previous running sum (3), giving us 4. So runningSum[1] is 3 + 1 which equals
- 4. 3. Next, the third element is 2. Adding this to the running sum we have so far (4), we get 6. Now, runningSum[2] is 4 + 2 which
- equals 6. 4. Finally, take the fourth element, 10. We add this to the last running sum (6), which gives us 16. Therefore, runningSum[3] is 6 +
- The final output, our running sum array, is [3, 4, 6, 16].

Implementing this in Python code without the use of accumulate() from the itertools module would look like this: 1 class Solution:

def runningSum(self, nums): for i in range(1, len(nums)): nums[i] += nums[i-1]

```
If we use this code with our example nums array [3, 1, 2, 10], here's what happens step by step in the loop:
  • i = 1: nums [1] (which is 1) is updated to nums [1] + nums [0] (which is 1 + 3), so nums [1] becomes 4.
```

return nums

• i = 2: nums[2] (which is 2) is updated to nums[2] + nums[1] (which is 2 + 4), so nums[2] becomes 6. • i = 3: nums[3] (which is 10) is updated to nums[3] + nums[2] (which is 10 + 6), so nums[3] becomes 16.

Python Solution from itertools import accumulate # Import accumulate function from itertools module

// Update the current element by adding the previous element's sum to it

// Loop through the array starting from the second element

for (int index = 1; index < nums.length; index++) {</pre>

// Return the modified array containing the running sum

nums[index] += nums[index - 1];

for (size_t i = 1; i < nums.size(); ++i) {</pre>

// Return the modified vector with the running sum

nums[i] += nums[i - 1];

// Add the previous element's value to the current element

And we get the final updated nums array [3, 4, 6, 16], which is the same result we calculated manually.

def runningSum(self, nums: List[int]) -> List[int]: # Calculate the running sum of the list of numbers using accumulate running_sum = list(accumulate(nums)) # Return the running sum as a list

class Solution:

```
return running_sum
Java Solution
   class Solution {
       // Method to calculate the running sum of the given array
       public int[] runningSum(int[] nums) {
```

12 13 } 14

```
11
           return nums;
C++ Solution
   #include <vector> // Include vector library for using the vector class
   class Solution {
   public:
       // Function to calculate the running sum of a 1D vector
       vector<int> runningSum(vector<int>& nums) {
           // Iterate over the vector starting from the second element
```

14 15 }; 16

return nums;

13

19

```
Typescript Solution
    * Computes the running sum of an array of numbers.
    * For every index in the array, it modifies the value at that index
    * to be the sum of all elements up to and including that index.
    * @param {number[]} numbers - The array of numbers to calculate the running sum for.
    * @returns {number[]} - The array of numbers with the running sum computed.
    */
   function runningSum(numbers: number[]): number[] {
       // Start iterating from the first index because the running sum at index 0 is just the element itself
       for (let index = 1; index < numbers.length; ++index) {</pre>
11
           // Update the element at the current index to be the sum of the element at the current index
13
           // and the element at the previous index which now contains the running sum up to the previous index
14
           numbers[index] += numbers[index - 1];
```

// Return the modified array with the running sums return numbers;

Time and Space Complexity

The time complexity of the code is O(n), where n is the length of the input array. This is because accumulate function goes through

each element of the array only once to calculate the running sum. The space complexity in the given code should be 0(n), not 0(1) as the reference answer suggests. This difference arises because although accumulate itself may operate with 0(1) additional space, calling list(accumulate(nums)) stores the results of

accumulate(nums) in a new list, which takes up O(n) space based on the number of elements in nums.