

# 303. Range Sum Query - Immutable

EasyDesignArrayPrefix Sum

## Problem Description

The LeetCode problem presents a common scenario in data processing — computing the sum of a subarray, which is a contiguous segment of an array. You are provided with an integer array `nums` and are expected to handle multiple queries asking for the sum of elements between two indices, `left` and `right` (both inclusive). To efficiently answer these queries, a data structure or algorithm is needed that can quickly calculate the sum of any given range in `nums`.

## Intuition

For solving this problem, a key observation is that repeatedly computing the sum of a range of elements directly from the array can be time-consuming, especially if the array is large or if there are many queries. To optimize this, a common approach is to use a technique called [prefix sum](#).

The [prefix sum](#) array is an auxiliary array where each element at index `i` stores the sum of all elements from the start of the original array up to index `i`. By preprocessing the input array into a prefix sum array, we can calculate the sum of any subarray in constant time. The sum of elements between indices `left` and `right` can be found by subtracting the prefix sum at `left - 1` from the prefix sum at `right`. This works because the prefix sum at `right` includes the total sum up to `right`, and if we subtract the sum up to `left - 1`, we are left with the sum from `left` to `right`.

In this solution, Python's `accumulate` function from the `itertools` module is used to create the [prefix sum](#) array easily. This function takes an iterable, in this case, `nums`, and returns a new iterable yielding accumulated sums. The additional `initial=0` parameter ensures that the `0`th index of the resulting prefix sum array (`self.s`) is `0`, which is helpful for handling cases where `left` is `0`.

By preparing this [prefix sum](#) array (`self.s`) during the initialization of the `NumArray` class, we ensure that each `sumRange` query can be answered in constant time by simply calculating `self.s[right + 1] - self.s[left]`, leading to an efficient solution for the problem at hand.

## Solution Approach

The provided Python code implements an efficient solution to the subarray sum problem by using the [prefix sum](#) technique. The prefix sum array is a powerful tool in algorithm design to solve range sum queries, reducing time complexity from potentially  $O(n)$  per query to  $O(1)$  per query after an initial preprocessing step.

Here's a step-by-step explanation of the code:

### Class Definition:

- `NumArray` is a class that takes an array and processes it to potentially answer many range sum queries.

### `__init__` Method:

- `self.s`: An instance variable that holds the [prefix sum](#) array.
- `accumulate(nums, initial=0)`: A call to Python's `accumulate` function, which constructs the prefix sum array from the input `nums`.
  - The `accumulate` function takes an iterable and returns an iterable with the accumulated values.
  - The `initial=0` parameter is important as it prefixes the resulted iterable with `0`, giving us the flexibility to handle the `sumRange` query accurately even when the left index is `0`.

### `sumRange` Method:

- The `sumRange` function computes the sum of elements in the range `[left, right]` by returning `self.s[right + 1] - self.s[left]`.
  - The reason for `right + 1` is because the [prefix sum](#) array is one element longer than the original array (`initial=0` has been added at the start), and sums are stored at one index ahead.

By using a [prefix sum](#) array, we trade off some space ( $O(n)$  additional space for the auxiliary array) and preprocessing time ( $O(n)$  time to construct the prefix sum array) for a massive gain in query time, reducing it to  $O(1)$  per query.

### Algorithm:

- Compute the prefix sums of the input array `nums` and store it in `self.s`.
- When the `sumRange` is called with `left` and `right` indices, return the sum for the specific range by the difference of prefix sums, which represents the sum of elements inclusively between `left` and `right`.

### Data Structures:

- A list `self.s`, which is essentially the auxiliary [prefix sum](#) array.

Using these concepts, the class `NumArray` allows for the fast computation of any given `sumRange` query, which is particularly useful for scenarios where there will be a large number of these queries on a pre-defined array where the contents do not change.

In summary, the implementation uses the [prefix sum](#) pattern to initialize a structure with  $O(n)$  complexity, but then allows each sum query to be answered in  $O(1)$  time, highlighting an effective trade-off for query-intensive use cases.

## Example Walkthrough

Here is a small example to illustrate the solution approach using a hypothetical array and a few queries:

Let's consider the following array: `nums = [3, 0, 1, 4, 2]`

- We initiate our `NumArray` object with this array which triggers the creation of the prefix sum array (`self.s`). The `accumulate` function cumulatively adds up each value in `nums` while including an initial `0` at the start. The resulting prefix sum array would look like:

```
self.s: [0, 3, 3, 4, 8, 10]
```

Explanation:

- Index 0: Initial value, `0`.
- Index 1: Sum up to `nums[0]` which is `3` (`0+3`).
- Index 2: Sum up to `nums[1]` which is `3` (`3+0` since `nums[1]` is `0`).
- Index 3: Sum up to `nums[2]` which is `4` (`3+1`).
- Index 4: Sum up to `nums[3]` which is `8` (`4+4`).
- Index 5: Sum up to `nums[4]` which is `10` (`8+2`).

- Suppose we want to know the sum from index `1` to `3` in the `nums` array. We use the `sumRange` method and provide the indices to it:

```
sumRange(1, 3) should return 0 + 1 + 4 = 5.
```

Using the prefix sum array `self.s`:

- We take the value at `right + 1` which is `self.s[3 + 1] = 8`
- We subtract the value at `left` which is `self.s[1] = 3`
- The result is `8 - 3 = 5`, which matches the expected output.

- Let's say we have another query asking for the sum from the start up to index `2`, that's `sumRange(0, 2)`:

```
sumRange(0, 2) should return 3 + 0 + 1 = 4.
```

Using the prefix sum method:

- Value at `right + 1` is `self.s[2 + 1] = 4`
- Value at `left` is `self.s[0] = 0` (since `left` is `0`, it naturally includes no numbers)
- The result is `4 - 0 = 4`, as expected.

These examples demonstrate how by initializing the prefix sum array once, we're able to answer multiple `sumRange` queries efficiently, each in constant time, without the need to re-calculate sums directly from the `nums` array. This becomes particularly powerful when dealing with a high volume of sum range queries on an unchanging array.

## Python Solution

```
1 from itertools import accumulate
2
3 class NumArray:
4     def __init__(self, nums: List[int]):
5         # Pre-calculate the cumulative sum of the array.
6         # The 'initial=0' makes sure the sum starts from index 0 for easier calculations.
7         self.cumulative_sum = list(accumulate(nums, initial=0))
8
9     def sumRange(self, left: int, right: int) -> int:
10        # Calculate the sum of elements between 'left' and 'right'
11        # by subtracting the sum up to 'left' from the sum up to 'right + 1'.
12        return self.cumulative_sum[right + 1] - self.cumulative_sum[left]
13
14
15 # Example of usage:
16 # numArray = NumArray(nums)
17 # sum = numArray.sumRange(left, right)
18
```

## Java Solution

```
1 class NumArray {
2     // The sum array stores the cumulative sum from the beginning up to the current index.
3     private int[] sumArray;
4
5     // Constructor that computes the cumulative sum of the numbers array.
6     public NumArray(int[] nums) {
7         int n = nums.length;
8         sumArray = new int[n + 1]; // Initialized with an extra element to handle the sum from 0 to ith index.
9
10        // Accumulate the sum of elements so that sumArray[i] holds the sum up to nums[i-1].
11        for (int i = 0; i < n; i++) {
12            sumArray[i + 1] = sumArray[i] + nums[i];
13        }
14    }
15
16    // Method to compute sum of elements within the range [left, right] both inclusive.
17    public int sumRange(int left, int right) {
18        // The sum of elements in range [left, right] is computed by subtracting the cumulative sum up to 'left' from the sum up to '
19        return sumArray[right + 1] - sumArray[left];
20    }
21 }
22
23 /**
24  * Usage example:
25  *
26  * NumArray obj = new NumArray(nums);
27  * int sum = obj.sumRange(left, right);
28  */
29
```

## C++ Solution

```
1 #include <vector>
2
3 class NumArray {
4 private:
5     // Prefix sum array to store the accumulated sum from the beginning up to each index.
6     std::vector<int> prefixSum;
7
8 public:
9     // Constructor that initializes the prefix sum array using the input 'nums' array.
10    NumArray(std::vector<int>& nums) {
11        int size = nums.size();
12        prefixSum.resize(size + 1); // Resizing with an extra element to handle the zero prefix sum.
13        prefixSum[0] = 0; // Initialize the zero-th index with 0 for the prefix sum.
14
15        // Calculate the prefix sum by adding the current element to the accumulated sum.
16        for (int i = 0; i < size; ++i) {
17            prefixSum[i + 1] = prefixSum[i] + nums[i];
18        }
19    }
20
21    // Function to calculate the sum of the elements in the range [left, right] in the 'nums' array.
22    int sumRange(int left, int right) {
23        // Return the difference between the prefix sums to get the range sum.
24        return prefixSum[right + 1] - prefixSum[left];
25    }
26 };
27
28 /**
29  * Usage:
30  * std::vector<int> nums = { ... };
31  * NumArray obj = new NumArray(nums);
32  * int sum = obj->sumRange(left, right);
33  * ...
34  * delete obj; // Don't forget to deallocate the memory when done.
35  */
36
```

## Typescript Solution

```
1 // Global variable to store the sum of elements up to each index.
2 let sumArray: number[] = [];
3
4 /**
5  * Initialize the sumArray with the prefix sum of the given nums array.
6  * @param nums - The input array of numbers.
7  */
8 function createNumArray(nums: number[]): void {
9     const n = nums.length;
10    sumArray = new Array(n + 1).fill(0);
11    for (let i = 0; i < n; ++i) {
12        sumArray[i + 1] = sumArray[i] + nums[i];
13    }
14 }
15
16 /**
17  * Calculates the sum of elements within the range [left, right] in the array.
18  * @param left - The starting index of the range (inclusive).
19  * @param right - The ending index of the range (inclusive).
20  * @returns The sum of elements within the range [left, right].
21  */
22 function sumRange(left: number, right: number): number {
23     return sumArray[right + 1] - sumArray[left];
24 }
25
26 // Example of usage:
27 // createNumArray([1, 2, 3, 4]);
28 // console.log(sumRange(1, 3)); // Output would be 9, which is the sum of elements [2, 3, 4].
29
```

## Time and Space Complexity

The provided code implements a class `NumArray` that precomputes the cumulative sum of an array to efficiently find the sum of elements in a given range.

### Time Complexity

- `__init__` Method:** The initial sum computation is done with `accumulate`, which processes each element once to create a cumulative sum list. This operation has a time complexity of  $O(n)$ , where `n` is the number of elements in the list `nums`.
- `sumRange` Method:** This method computes the sum in constant time by subtracting the accumulated sum at the `left` index from the accumulated sum at the `right + 1` index. The time complexity for each `sumRange` query is  $O(1)$ .

### Space Complexity

- The space complexity of the `NumArray` class is primarily determined by the cumulative sum list `self.s`. Since this list has one more element than the input list (due to `initial=0`), the space complexity is  $O(n)$ , where `n` is the number of elements in the input list `nums`.

Overall, the preprocessing step (`__init__` method) requires  $O(n)$  time, and each `sumRange` query can be answered in  $O(1)$  time, with a space complexity of  $O(n)$ .