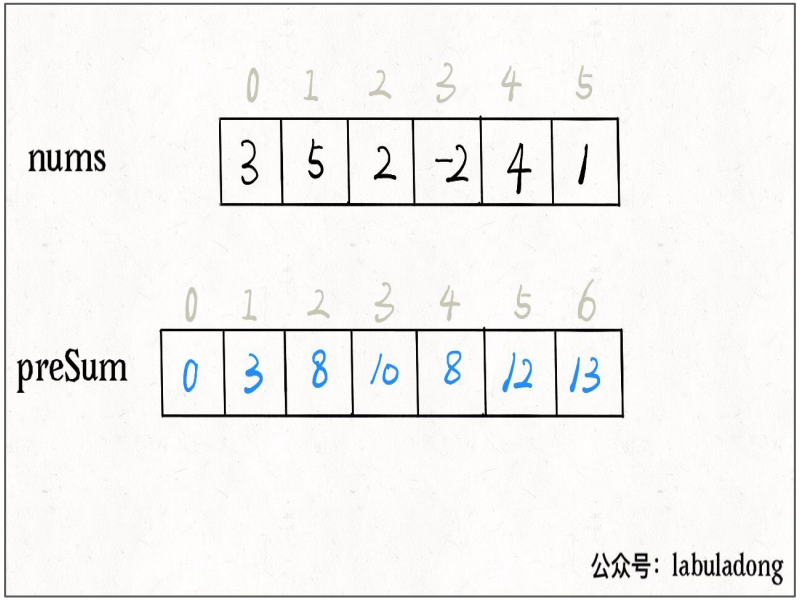
Prefix Sum



Lecture Flow for Prefix Sum

1) Pre-requisites
2) Definition
3) Examples and Algorithms
4) Variants
5) Pitfalls
6) Complexity Analysis
7) Applications of prefix sum
8) Practice Questions

Pre-requisites

- Arrays
- Familiarity with arithmetic operations
- Understanding pointers or indices

Definition

- Prefix sum, also known as cumulative sum, is a technique used to efficiently answer range sum queries on an array of numbers. It involves precomputing the sum of elements from the start of the array up to each index and storing the results in a separate array.

Cont.

- To calculate the **prefix sum**,
  - Grab the previous value of the prefix sum.
  - Add the current value of the traversed array.

Variants

There are several variants on Prefix Sum:

1. 1D Prefix Sum
2. 2D Prefix Sum
3. Inclusive Prefix Sum
4. Exclusive Prefix Sum
5. Running Sum
6. Range updates

1D Prefix Sum

- The 1-dimensional prefix sum algorithm is a simple and efficient technique for calculating the cumulative sum of a sequence of numbers.
- The basic idea is to iterate over the sequence and compute the sum of all previous elements for each element in the sequence.
- The resulting array is called the prefix sum array.

1D Prefix Sum

- Here's an example to illustrate the 1-dimensional prefix sum algorithm:

  numbers: [3, 1, 7, 0, 4, 1, 6]

  Prefix_sum: [3, 4, 11, 11, 15, 16, 22]

Pair Programming

Problem Link

Problem Pattern

Given an integer array `nums`, handle multiple queries of the following type:

1. Calculate the sum of the elements of `nums` between indices `left` and `right` inclusive where `left <= right`.

Implement the `NumArray` class:

- `NumArray(int[] nums)` Initializes the object with the integer array `nums`.
- `int sumRange(int left, int right)` Returns the sum of the elements of `nums` between indices `left` and `right` inclusive (i.e. `nums[left] + nums[left + 1] + ... + nums[right]`).

Example:

Input: ["NumArray", "sumRange", "sumRange", "sumRange"]

[[[-2, 0, 3, -5, 2, -1]], [0, 2], [2, 5], [0, 5]]

Output: [null, 1, -1, -3]

Implementation

```python
class NumArray:
    def __init__(self, nums: List[int]):
        self.prefix_sum = [0]
        for num in nums:
            self.prefix_sum.append(self.prefix_sum[-1] + num)

    def sumRange(self, left: int, right: int) -> int:
        leftSum = self.prefix_sum[left]
        rightSum = self.prefix_sum[right+1]

        return rightSum - leftSum
```

Problem Pattern

Given an array of integers nums, calculate the **pivot index** of this array.
The **pivot index** is the index where the sum of all the numbers **strictly** to the left of the index is equal to the sum of all the numbers **strictly** to the index's right.

If the index is on the left edge of the array, then the left sum is 0 because there are no elements to the left. This also applies to the right edge of the array.

Return *the* ***leftmost pivot index***. If no such index exists, return -1.

Input: nums = [1,7,3,6,5,6]

Output: 3

Explanation: The pivot index is 3.

Left sum = nums[0] + nums[1] + nums[2] = 1 + 7 + 3 = 11

Right sum = nums[4] + nums[5] = 5 + 6 = 11

2D Prefix Sum

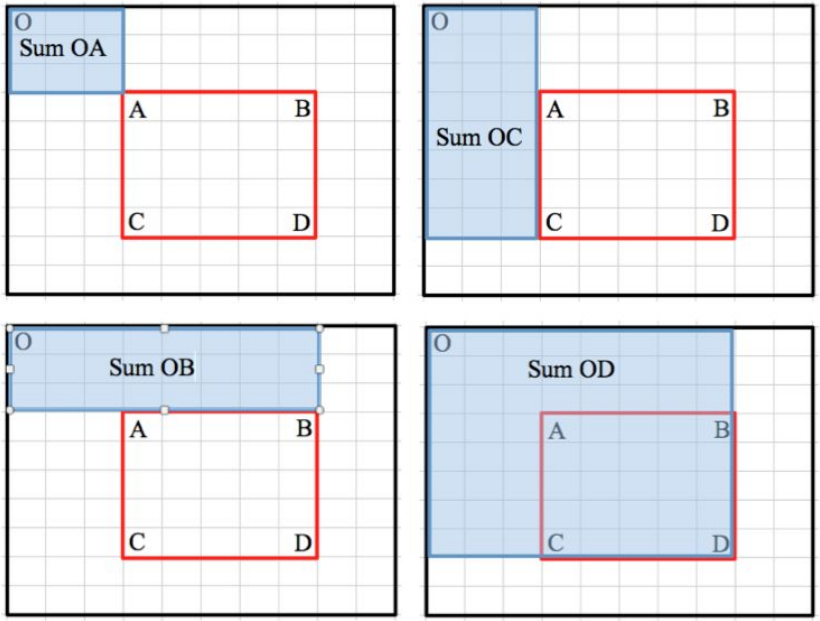
- The 2D prefix sum is a technique used to efficiently calculate the sum of a sub-rectangle in a 2D array.
- The basic idea is to precompute the sum of all elements in the rectangle with upper left corner (0, 0) and lower right corner (i, j), where i and j are the row and column indices, respectively.

2D Prefix Sum

- This can be done using the 2D prefix sum array, which stores the cumulative sum of elements from (0, 0) to (i, j) for all i and j.

  Formula: **prefix[i][j]=prefix[i−1][j]+prefix[i][j−1] − prefix[i−1][j−1]+arr[i][j]**

2D Prefix-Sum



$$Sum(ABCD) = Sum(OD) - Sum(OB) - Sum(OC) + Sum(OA)$$

Problem Pattern

Given a 2D matrix `matrix`, handle multiple queries of the following type:

- Calculate the sum of the elements of `matrix` inside the rectangle defined by its upper left corner `(row1, col1)` and lower right corner `(row2, col2)`.

Implement the `NumMatrix` class:

- `NumMatrix(int[][] matrix)` Initializes the object with the integer matrix `matrix`.
- `int sumRegion(int row1, int col1, int row2, int col2)` Returns the sum of the elements of `matrix` inside the rectangle defined by its upper left corner `(row1, col1)` and lower right corner `(row2, col2)`.

You must design an algorithm where `sumRegion` works on ***O(1)*** time complexity.

Pair Programming

Problem Link

Implementation

```
class NumMatrix:
    def __init__(self, matrix: List[List[int]]):
        rows, cols = len(matrix), len(matrix[0])

        # build prefix 2D prefix Table (PS)
        self.ps = [[0] * (cols + 1) for _ in range(rows + 1)]
        for r in range(rows):
            for c in range(cols):
                self.ps[r + 1][c + 1] = (self.ps[r + 1][c] + self.ps[r][c + 1] - self.ps[r][c] + matrix[r][c])

    def sumRegion(self, row1: int, col1: int, row2: int, col2: int) -> int:
            return (self.ps[row2 + 1][col2 + 1] - self.ps[row1][col2 + 1]
                        - self.ps[row2 + 1][col1] + self.ps[row1][col1])
```

Inclusive Prefix Sum

- Sums up all the elements before and **including the current element**
- Given an array of n elements

  A = $[a_0, a_1, a_2, a_3, \ldots, a_{n-1}]$

The inclusive prefix sum will be:

prefix_sum = $[a_0, a_0 + a_1, a_0 + a_1 + a_2, \ldots, a_0 + \ldots + a_{n-2} + a_{n-1}]$

Inclusive Prefix Sum Cont.

Example:

A = [3, 1, 7 , 0, 4, 1, 6, 3]

prefix_sum = [3, 4, 11, 11, 15, 16, 22, 25]

Implementation of Inclusive Prefix Sum

- For given an array called nums

```
# identity element
accumulator = 0

for i in range(len(nums)):
    accumulator += nums[i]
    nums[i] = accumulator
```

Exclusive Prefix Sum

- Sums up all the elements before and **not including** the current element
- Given an array of n elements
  $A = [a_0, a_1, a_2, a_3, \ldots, a_{n-1}]$

The **exclusive prefix** sum will be:

prefix_sum = $[i, i + a_0, a_0 + a_1, a_0 + a_1 + a_2, \ldots, a_0 + \ldots + a_{n-3} + a_{n-2}]$

Where i = is the identity / zero element

Exclusive Prefix Sum Cont.

Example:

A = [3, 1, 7, 0 , 4, 1, 6, 3]

prefix_sum = [0, 3, 4, 11, 11, 15, 16, 22]

Implementation of Exclusive Prefix Sum

- For given an array called nums

```
# identity element
accumulator = 0


for i in range(len(nums)):
    nums[i] = accumulator
    accumulator += nums[i]
```

Running Sum

- The **summation** of a sequence of numbers which is updated each time a new number is added to the sequence
  - done by adding the value of the new number to the previous running total.

- It allows the total to be stated at any point in time without having to sum the entire sequence each time
- It can save having to record the sequence itself, if the particular numbers are not individually important.

Problem Pattern

Example: consider the sequence <5, 8, 3, 2>

Total = 5 + 8 + 3 + 2

Now insert 6 to the sequence <5, 8, 3, 2, 6>

Total = 5 + 8 + 3 + 2 + 6

But if we regarded running_sum = 18 at first

now running_sum = 18 + 6 = 24

- we would not even need to know the sequence at all

Pair Programming

Problem Link

1D range updates using prefix sum

Motivating problem:

Array Manipulation

Problem Summary:

We are given an array and multiple queries asking us to add a constant value to all elements in the range l to r

Approach:

★ What if we were asked to add a constant value to the suffix of the array starting at index l, for multiple queries?

Example: given nums= [1, 3, 5, 7] and queries = [[1,2],[2,1]]

where queries[i] = [l,k] for: 0 < i < len(queries),

l index(0 - indexed),

k = constant value

- Instead of updating the whole suffix for each query, we can only increment the first index of the suffix for each query,

  i.e nums[l] += k.

  For i = 0: nums= [1,5,5,7]

  For i = 1: nums= [1,5,6,7]

- Then after all queries, we take the prefix sum of the whole array.

  nums= [1, 6, 12, 19]

★ Now, what about updating the range l to r ?

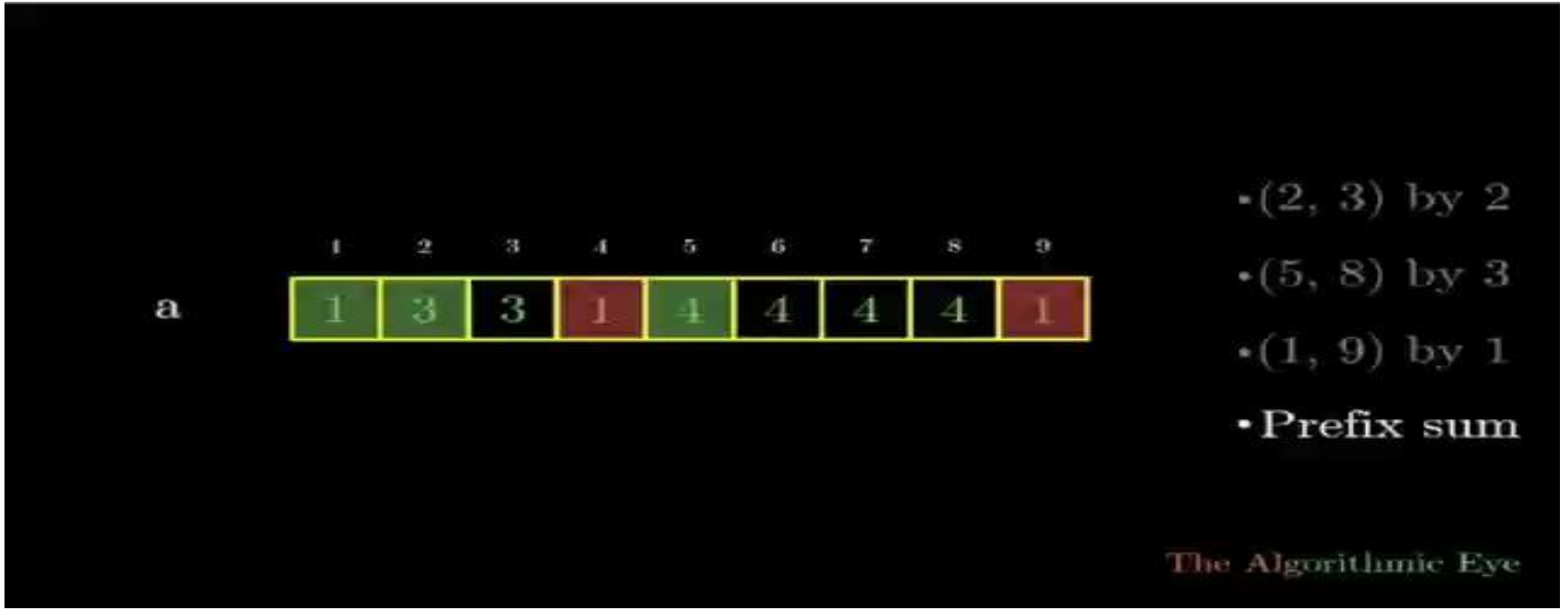
- you can increment the suffix starting at index l by k and decrement the suffix starting at index r+1 by k, for each query

  i.e nums[l] += k and

  nums[r+1] -= k

  then take prefix sum of the array as before.

A visual summary:



Pair Programming

Problem Link

Implementation

```
k = 1

# queries
for l,r in queries:
    nums[l] += k

    if r + 1 < len(nums):
        nums[r+1] -= k

//# take prefix sum
for i in range(1,len(nums)):
    nums[i] += nums[i-1]
```

Pitfalls

There are several Pitfalls on sliding window:

1. Off-by-one Errors
2. Inconsistent Indexing
3. Forgetting to initialize the Prefix Sum
4. Misunderstanding the Problem Requirements
5. Not considering edge cases

Off-by-one Errors

These errors occur when we incorrectly compute the indices or bounds of the array, resulting in incorrect or unexpected results.

For example, suppose we have an array A of size n and we want to compute the sum of elements in the range [l, r]. If we use the prefix sum technique, we can compute the sum of this range as sum[r] - sum[l-1], where sum[i] represents the cumulative sum of the first i elements of A.

Off-by-one Errors Cont.

original_arr = [1,2,3,4,5]

prefix_sum = [1,3,6,10,15]

Get the sum from index [0, 2] **?**

Inconsistent Indexing

This can happen when we are computing the prefix sum and using different indexing schemes for the array and the prefix sum.

For example, suppose we have an array A of size n and we want to compute the prefix sum array S, where S[i] represents the sum of the first i elements of A. If we use a 0-based indexing scheme for A but mistakenly use a 1-based indexing scheme for S, we may end up with an inconsistent prefix sum that doesn't match the original array.

Forgetting to initialize the Prefix Sum

This can result in incorrect or unexpected results when computing range sums or updates.

For example, suppose we have an array A of size n and we want to compute the prefix sum array S. If we forget to initialize S[0] to A[0], we may end up with incorrect prefix sums for the rest of the array.

Misunderstanding the Problem Requirements

Misunderstanding the Input Requirements:

Don’t assume that the input array must be sorted or that it must have certain properties

Misunderstanding the Output Requirements:

Don’t assume that the output array must be sorted, or that it must have a specific size or format.

Misunderstanding the Algorithm Requirements:

The optimal algorithm for prefix sum problems is to use a single pass over the input array to compute the prefix sums in linear time.

Misunderstanding the Problem Constraints:

It's essential to understand the problem constraints to ensure the solution is valid.

Not considering edge cases

Why bother?

To Avoid the following:

- Incorrect result
- Unexpected behaviour
- Performance issue
- Missed optimization opportunities
- Unhandled exceptions

Examples?

Edge cases to look out for while working with prefix sum questions:

- Empty array
- Single element array
- Array with negative elements
- Array with all negative elements
- Subarrays with first or last index of the input array
- Subarray of size 1
- Subarray with zero-sum
- Large input sizes

Complexity Analysis

- 1D Prefix Sum
  - Time complexity: **O(n)**
  - Space complexity: **O(n)/O(1)**
- 2D Prefix Sum
  - Time complexity: **O(n * m)**
  - Space complexity: **O(n * m)**
- 1D range updates for adding a constant to the range per query
  - Time complexity: **O(1)**
  - Space complexity: **O(n)/O(1)**

Applications of prefix sum

1. Computing subarray sums
2. Range updates and queries
3. Computing running averages
4. Counting occurrences
5. Dynamic programming

Practice Questions

- Range Sum Query - Immutable
- Subarray Sum Equals K
- Maximum Subarray
- Product of Array Except Self
- Count Number of Nice Subarrays
- Subarray Sums Divisible by K
- Number of Pairs
- Count Triplets That Can Form Two Arrays of Equal XOR
- Range Sum Query 2D - Immutable
- Car Pooling
- Maximum Sum Obtained of Any Permutation

Quote

"Believe you can and you're halfway there."

Theodore Roosevelt