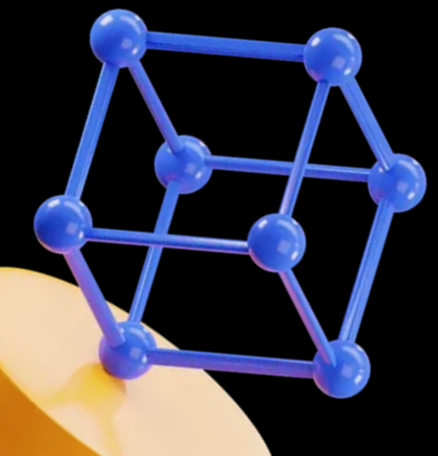
Sohum Sheth, Dean Richardson

HKD MEDIUM SECTION #1

Learning Prefix Sums

January 28th, 2026

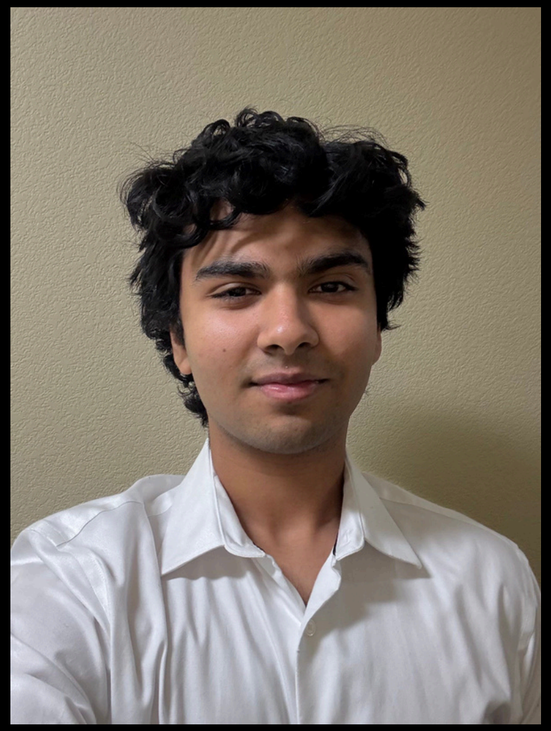


Attendance and Announcements



- 🍕Pizza at the end as always :)

Meet the Medium Officers!!!!



Sohum Sheth

Computer Systems Engineering

Likes lazy weekends with my dog



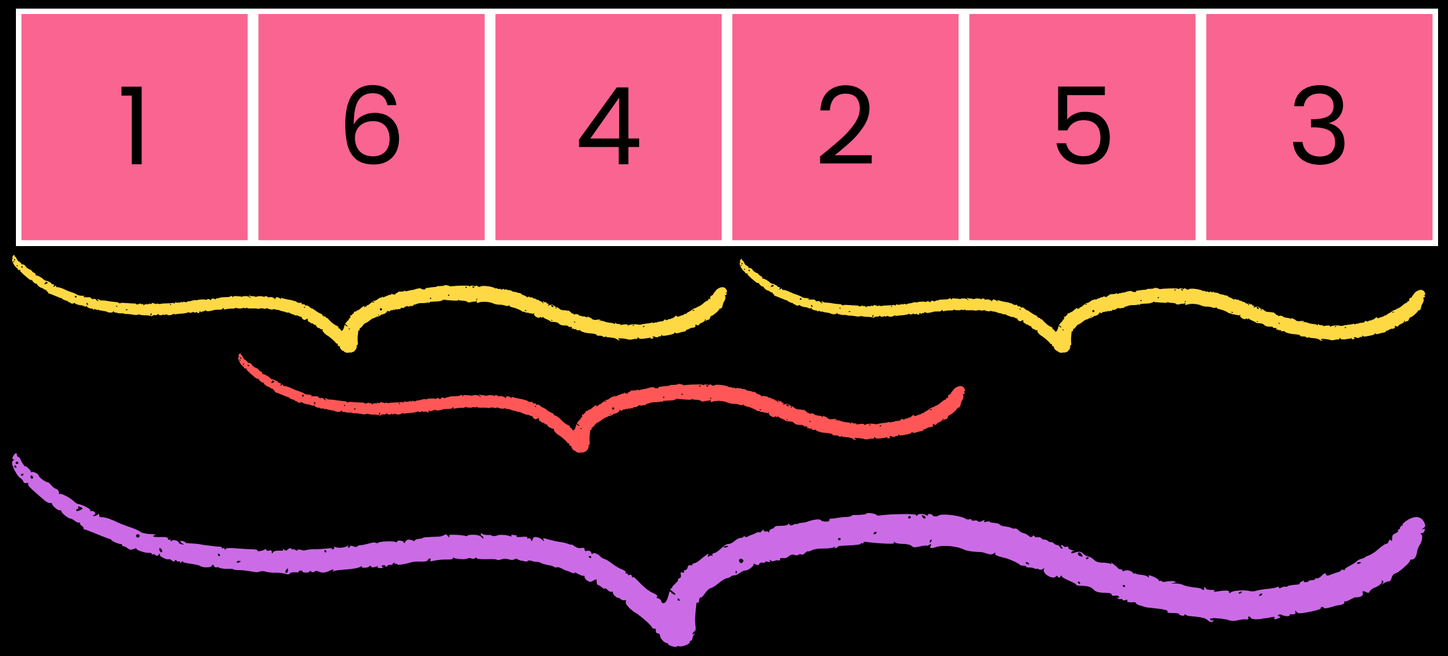
Dean Richardson

Computer Systems Engineering

Likes dogs and vintage tech

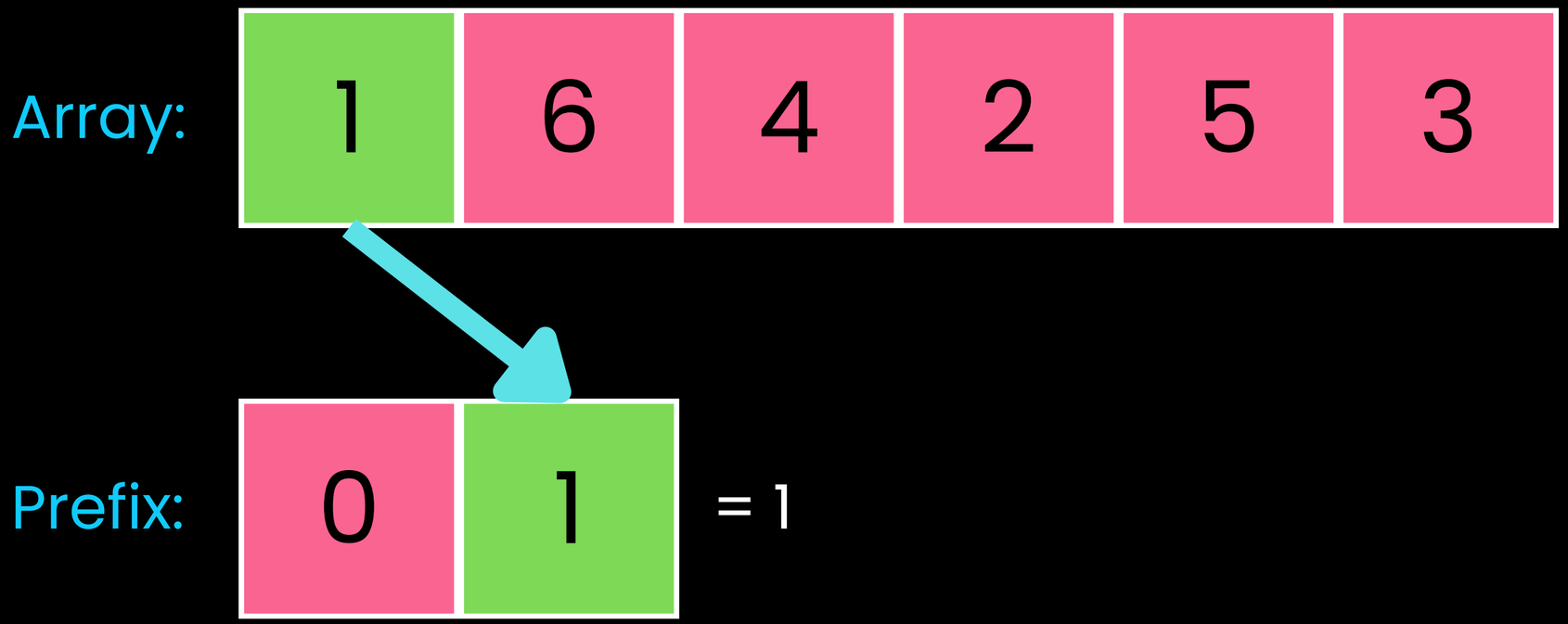
What are Prefix Sums?

Context: Imagine we have an array and we are tasked with finding the sum of all elements in the array across several ranges



What are Prefix Sums?

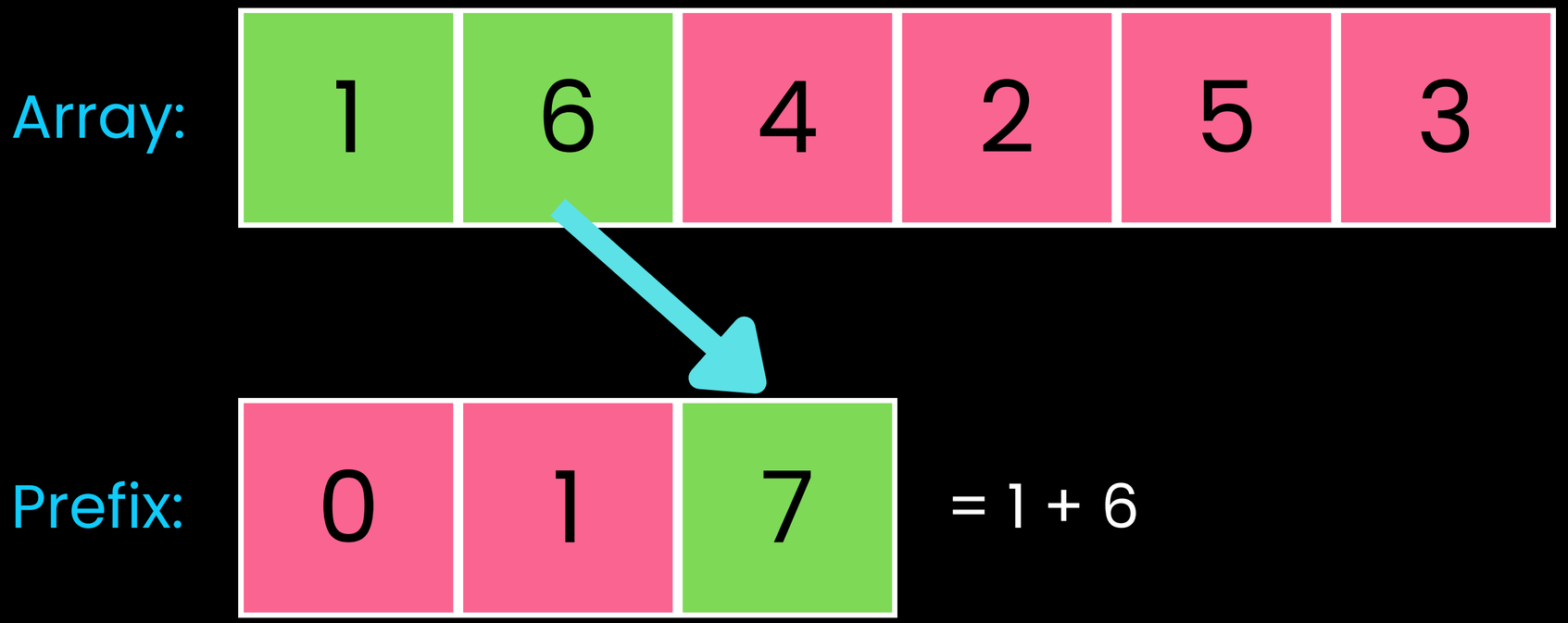
Solution: To reduce the number of calculations, we can set up a separate array that tracks the sum across the whole array!



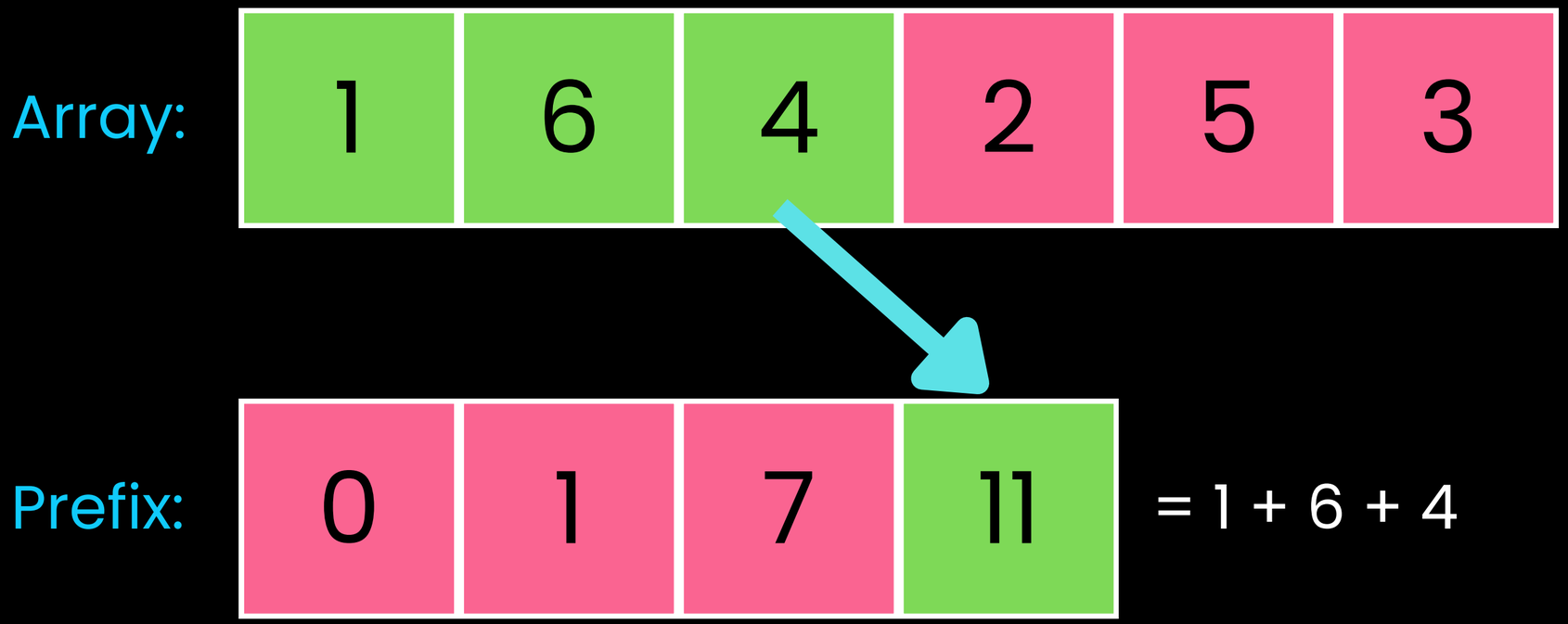
(We need to start this array with a zero, we'll explain later; keep it in the back of your mind!)

What are Prefix Sums?

Each element in the prefix array should equal the sum of all elements before it in the starting array

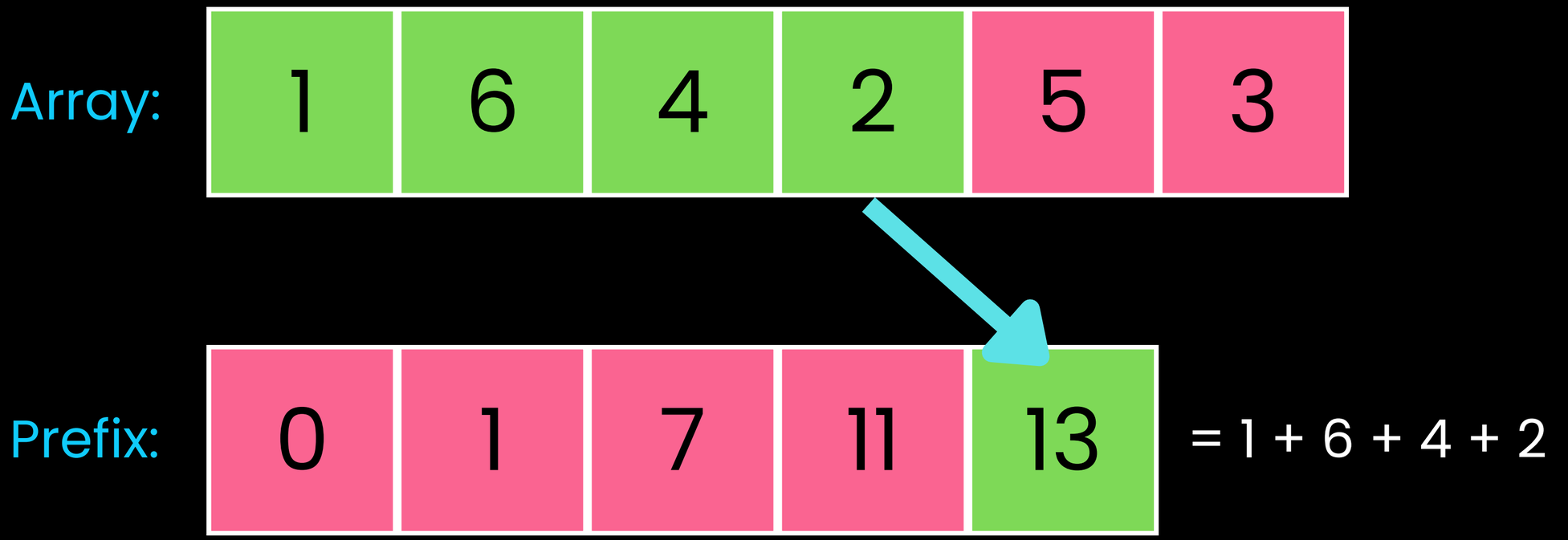


What are Prefix Sums?

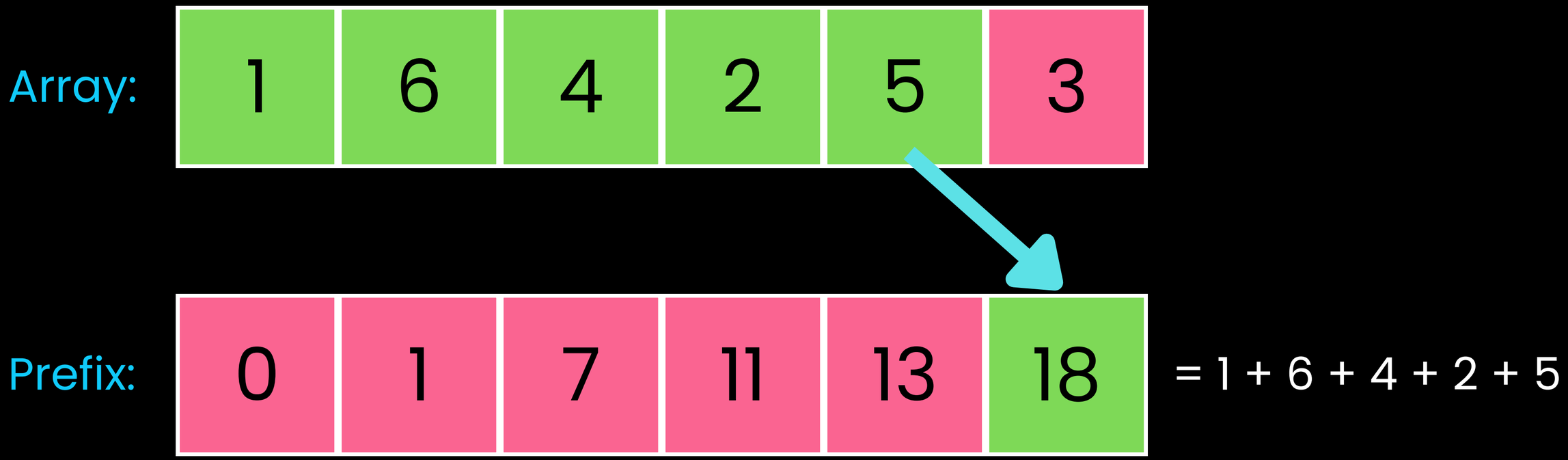


What are Prefix Sums?

We keep looping until we've added every element together

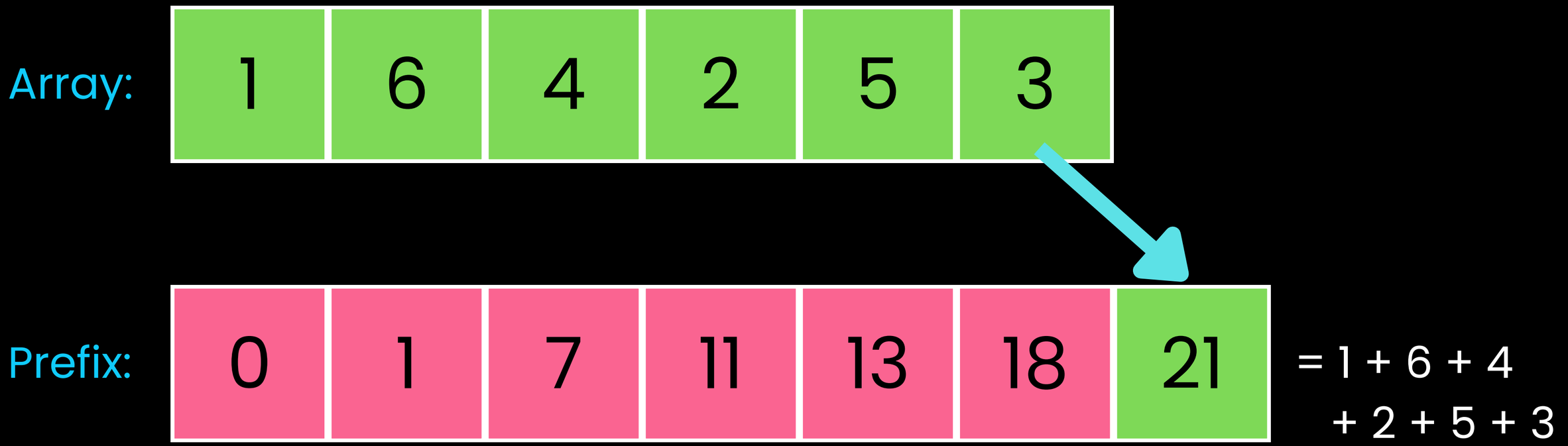


What are Prefix Sums?

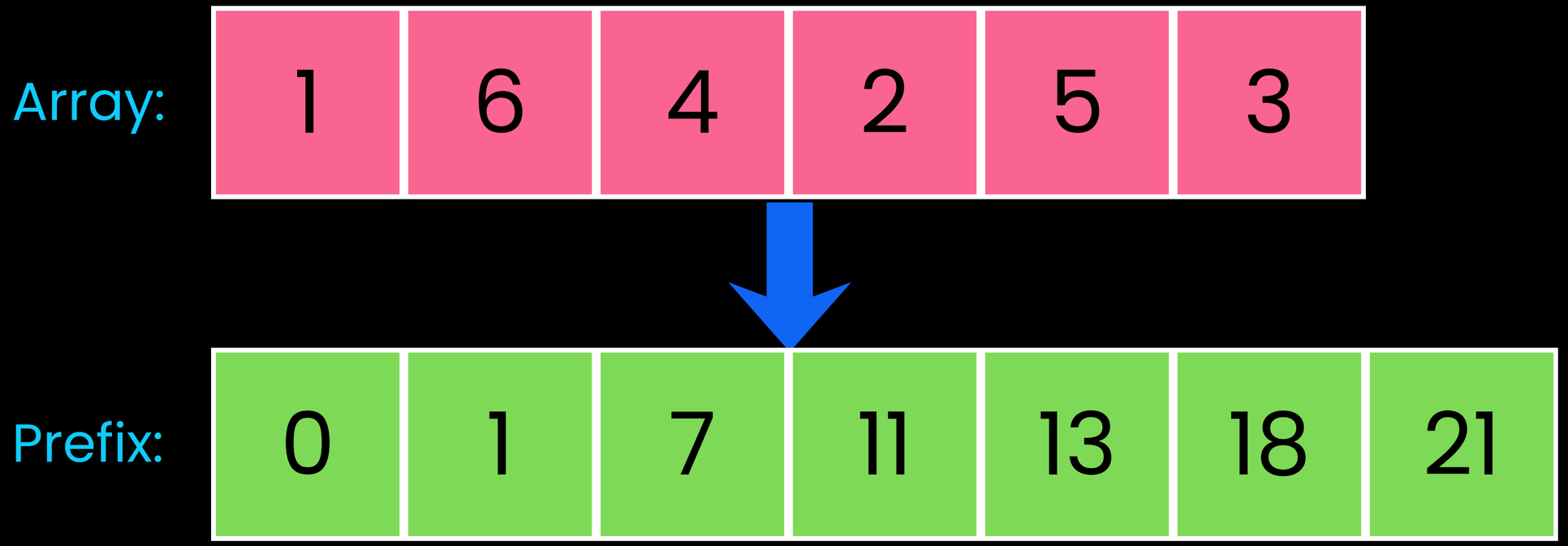


What are Prefix Sums?

Finished!

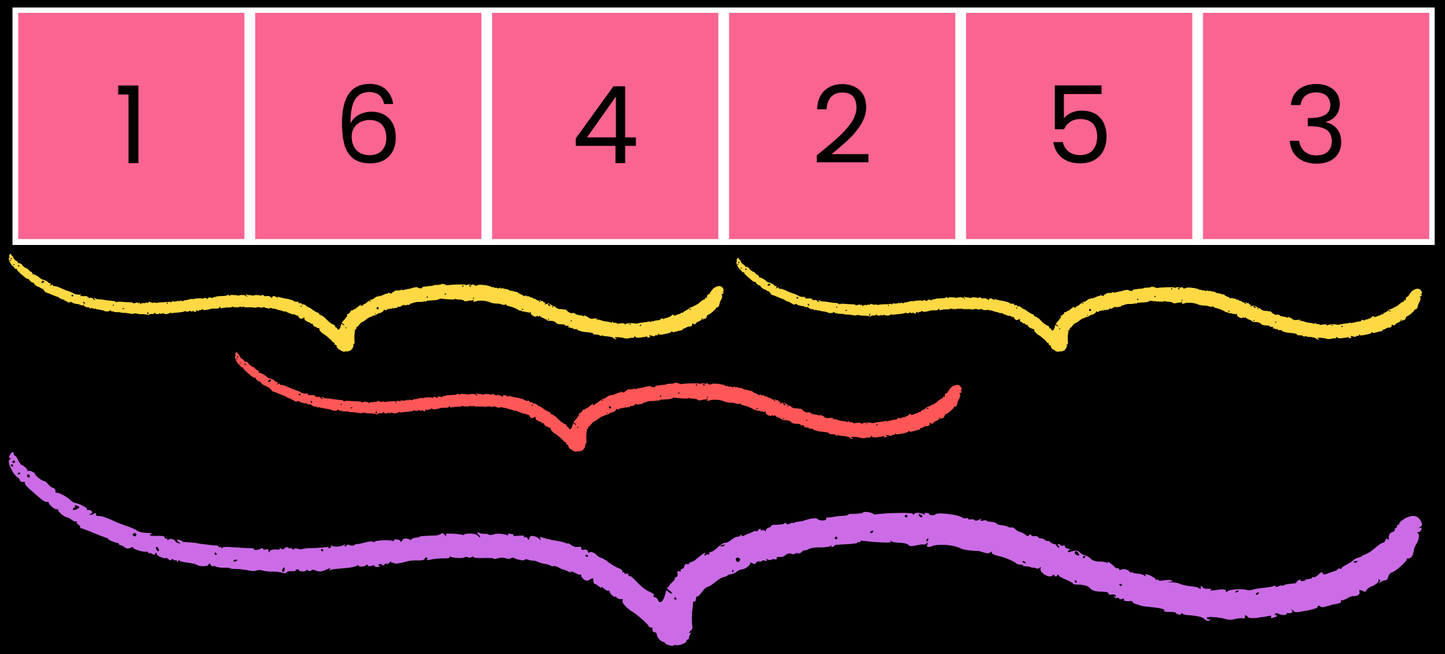


What are Prefix Sums?



Why?

Imagine we have an array and we are tasked with finding the sum of all elements in the array across several ranges



Let’s try using our new prefix array to speed this up!

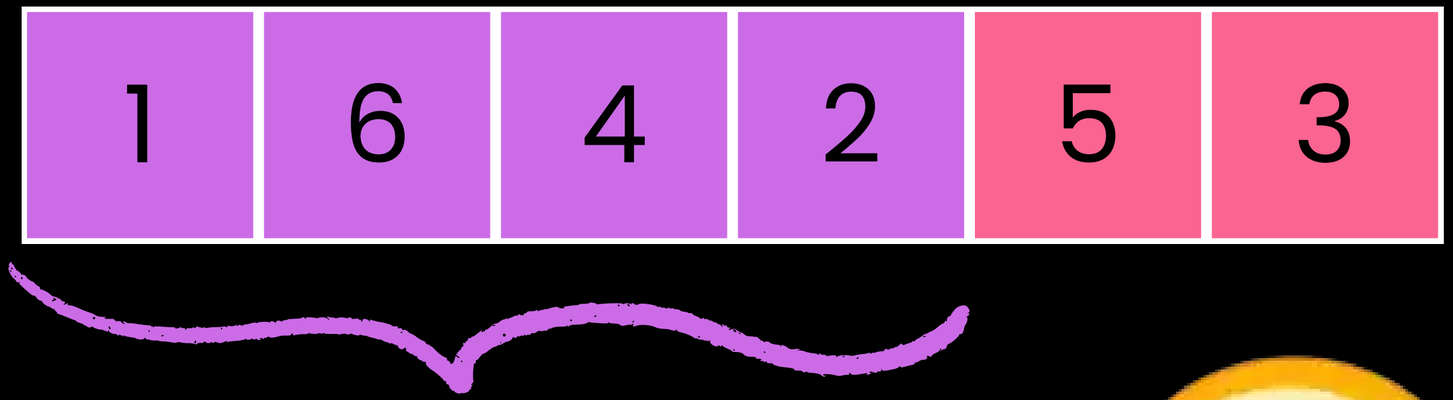
Problem

Let’s say that we want to calculate the sum of the first four elements of the array

Without Prefix Sums

Now, let's say that we want to calculate the sum of the first four elements of the array

Array:

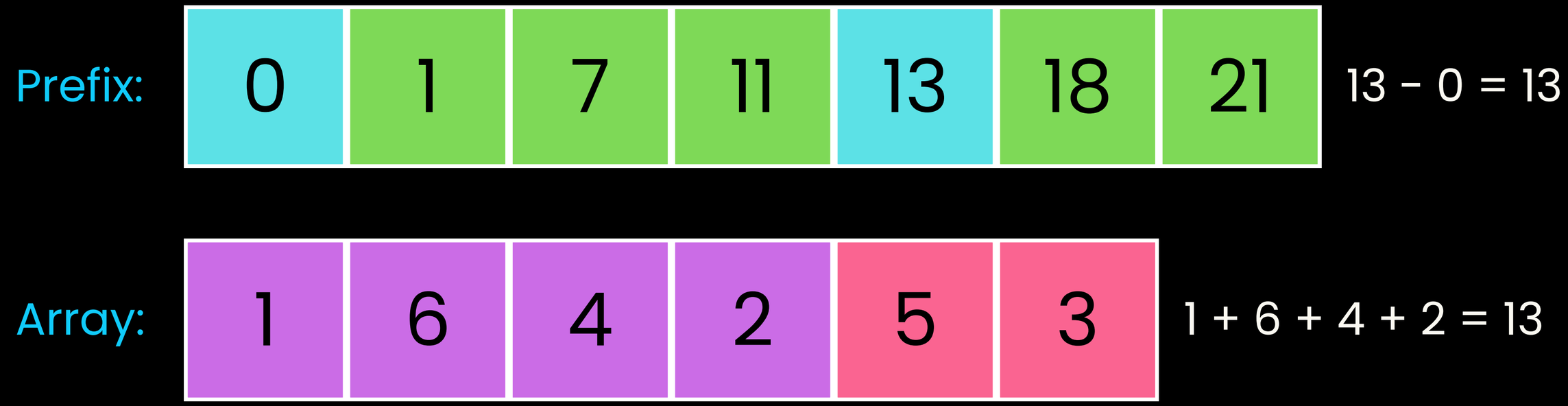


Sum = 1 + 6 + 4 + 2 = 13

3 Operations! Very Slow



With Prefix Sums



With prefix sums, instead of adding all the numbers together, we can just subtract two numbers to get the sum!

Sum = 13 - 0 = 0

This is also why we had to start the prefix array with 0!

New Challenge

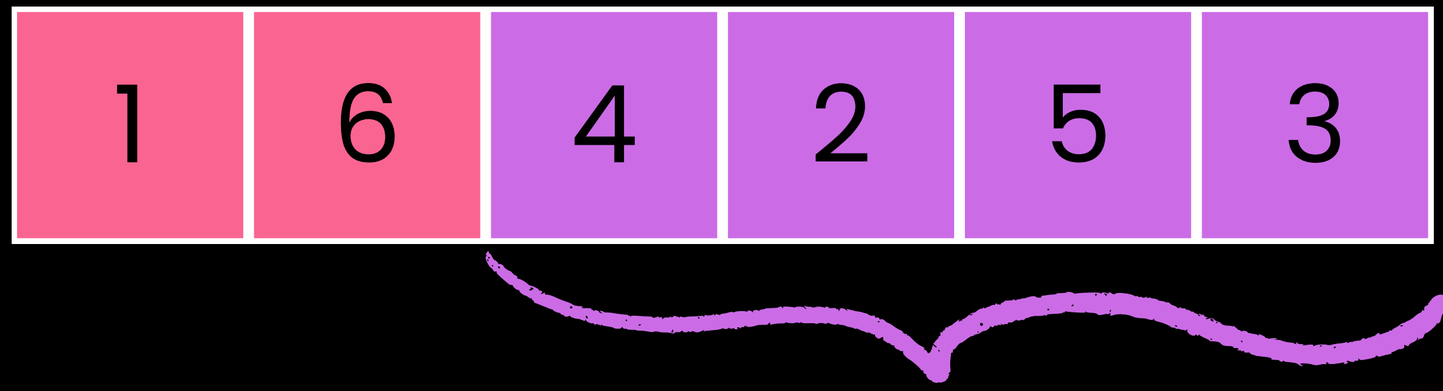
What if we want to calculate the sum of the last four elements?



Without Prefix Sums

Now, let's say that we want to calculate the sum of the last four elements of the array

Array:



Sum = 4 + 2 + 5 + 3 = 14

3 Operations! Very slow!



With Prefix Sums

Prefix:	0	1	7	11	13	18	21

21 - 7 = 14

Array:	1	6	4	2	5	3

4 + 2 + 5 + 3 = 14

Sum = 21 - 7 = 14

Only one subtraction needed!



The Warmup

https://leetcode.com/problems/range-sum-query-immutable

303. Range Sum Query - Immutable

Easy | Topics | Companies

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 1:

```
Input
["NumArray", "sumRange", "sumRange", "sumRange"]
[[[-2, 0, 3, -5, 2, -1]], [0, 2], [2, 5], [0, 5]]
Output
[null, 1, -1, -3]

Explanation
NumArray numArray = new NumArray([-2, 0, 3, -5, 2, -1]);
numArray.sumRange(0, 2); // return (-2) + 0 + 3 = 1
numArray.sumRange(2, 5); // return 3 + (-5) + 2 + (-1) = -1
numArray.sumRange(0, 5); // return (-2) + 0 + 3 + (-5) + 2 +
(-1) = -3
```

Constraints:

- `1 <= nums.length <= 10^4`
- `-10^5 <= nums[i] <= 10^5`
- `0 <= left <= right < nums.length`
- At most `10^4` calls will be made to `sumRange`.

What is the Problem Asking?

Our goal is to design a class with two functions:

- Function 1: Constructor
  - This function will accept and store an array of integers

- Function 2: sumRange
  - This function will take as input two numbers, corresponding to a range of indexes, such as (0,2) or (3,5)
  - The function will return the sum of all of the array's numbers in this range of indexes

Test Case

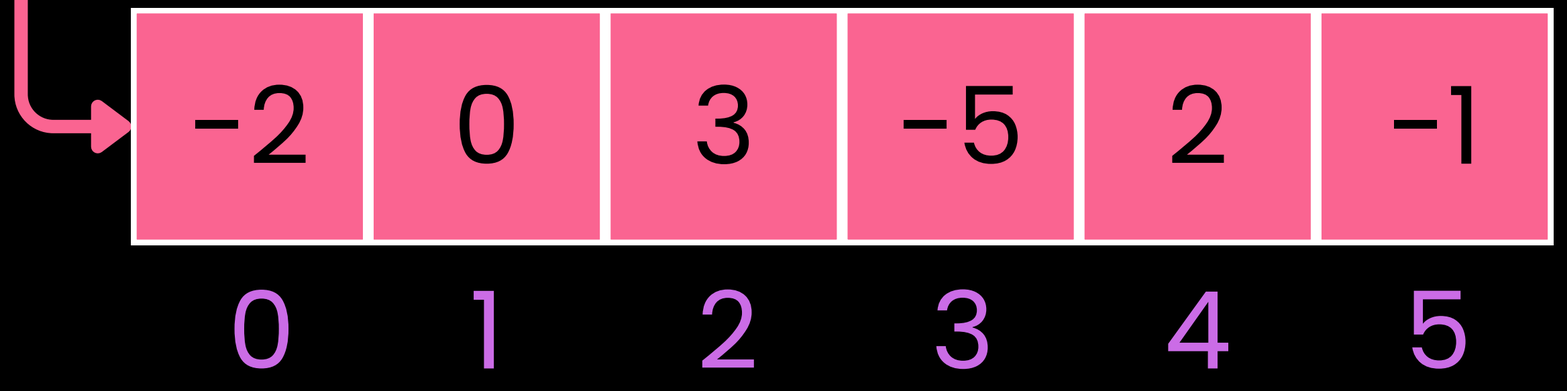
Example 1:

Input

```
["NumArray", "sumRange", "sumRange", "sumRange"]
[[[-2, 0, 3, -5, 2, -1]], [0, 2], [2, 5], [0, 5]]
```

Output

```
[null, 1, -1, -3]
```



Test Case

Example 1:

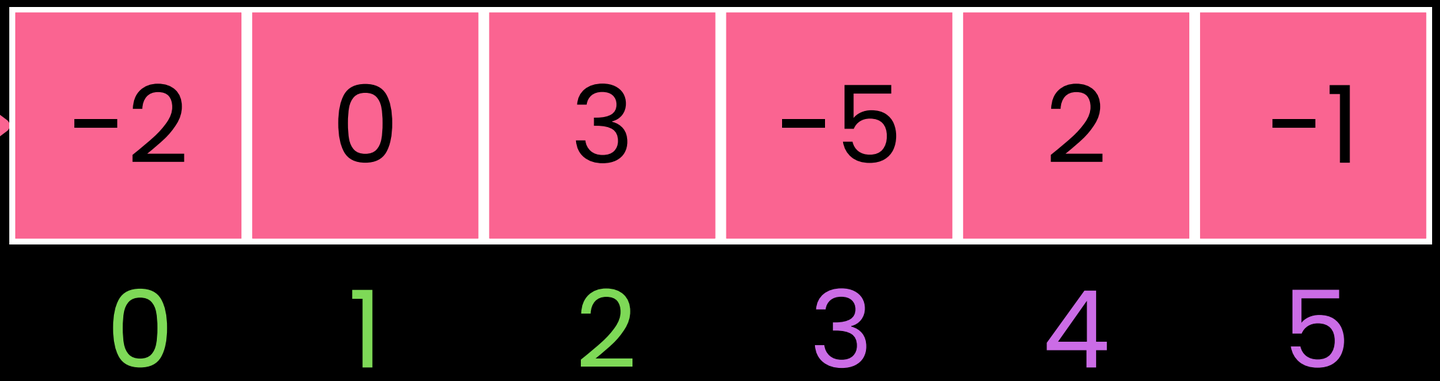
Input

```
["NumArray", "sumRange", "sumRange", "sumRange"]
[[[-2, 0, 3, -5, 2, -1]], [0, 2], [2, 5], [0, 5]]
```

Output

```
[null, 1, -1, -3]
```

Asking for the sum from indexes 0 through 2



Sum = -2 + 0 + 3

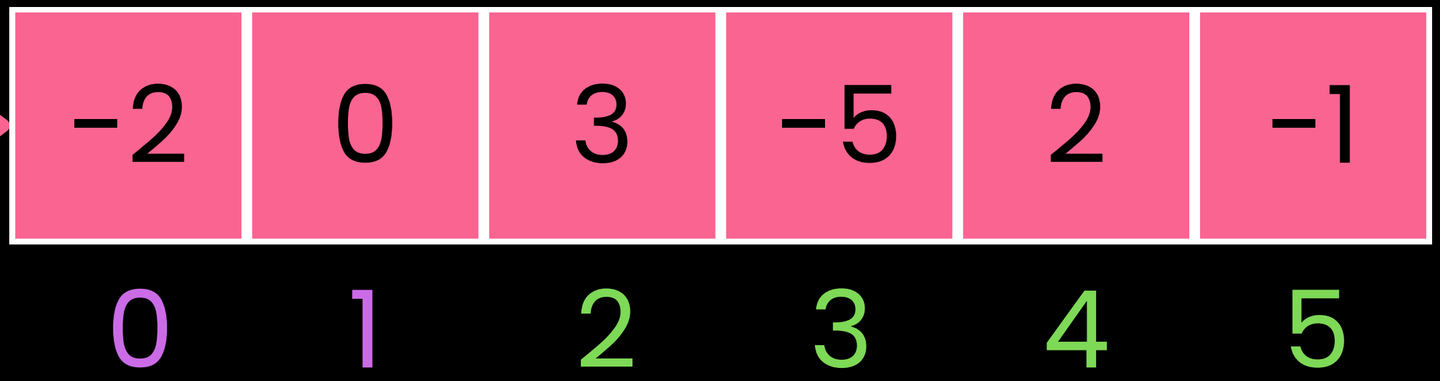
Sum = -1

Test Case

Example 1:

Input
```
["NumArray", "sumRange", "sumRange", "sumRange"]
[[[-2, 0, 3, -5, 2, -1]], [0, 2], [2, 5], [0, 5]]
```
Output
```
[null, 1, -1, -3]
```

Asking for the sum from indexes 2 through 5



Sum = 3 - 5 +2 - 1
Sum = -1

Test Case

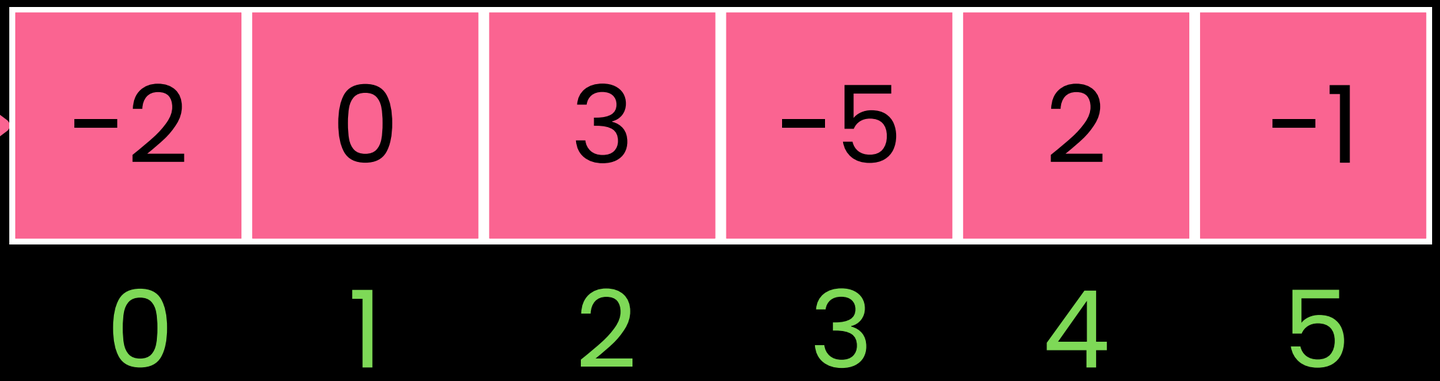
Example 1:

Input

```
["NumArray", "sumRange", "sumRange", "sumRange"]
[[[-2, 0, 3, -5, 2, -1]], [0, 2], [2, 5], [0, 5]]
```

Output

```
[null, 1, -1, -3]
```



Asking for the sum from indexes 0 through 5

Sum = -2 + 0 + 3 - 5 + 2 -1

Sum = -3

Final Notes

Remember, this is a **class** with **two separate functions**

Function 1 (Constructor):

- Takes the array of integers as input
- **If you would like to store any other variables inside the class, do it here**

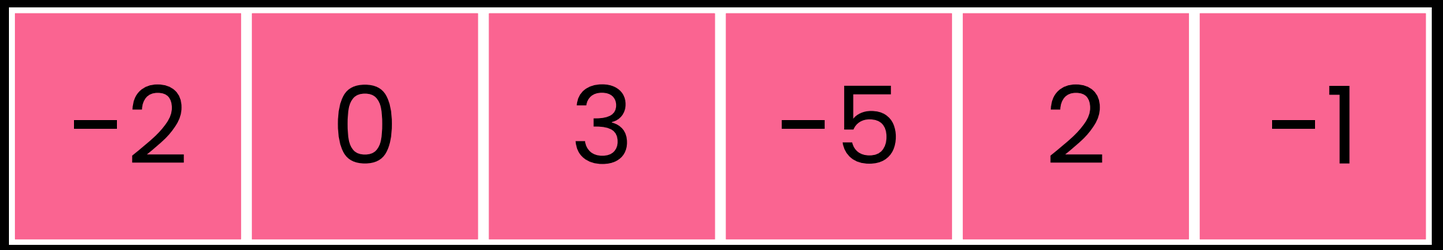
Function 2: sumArray:

- Accepts two integers corresponding to a range of indexes
- Returns the sum of all integers in the range

Solution

We can efficiently solve this problem by generating a prefix sum Array!

Let’s look at our test case again:



Solution

Let’s look at our test case again:

-2	0	3	-5	2	-1

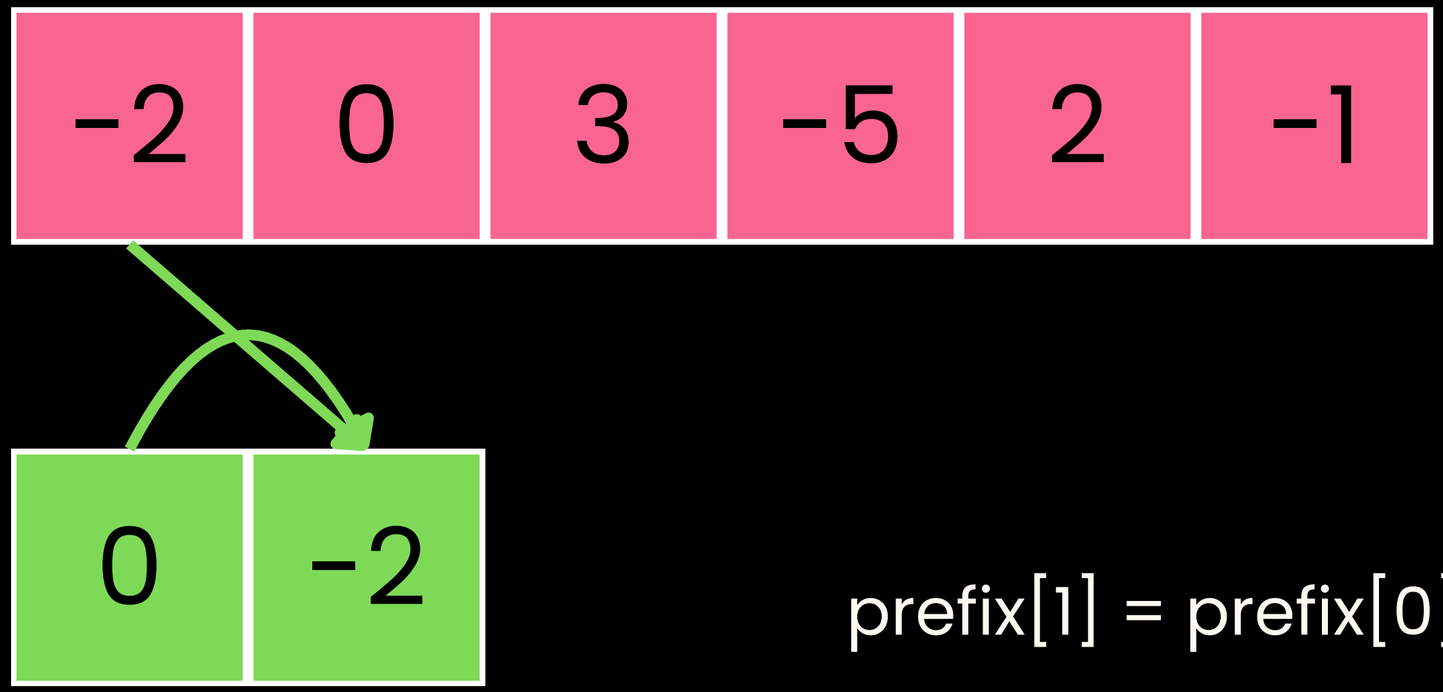
To start our prefix sum, we’ll make an array beginning with a zero

prefix[0] = 0

0

Solution

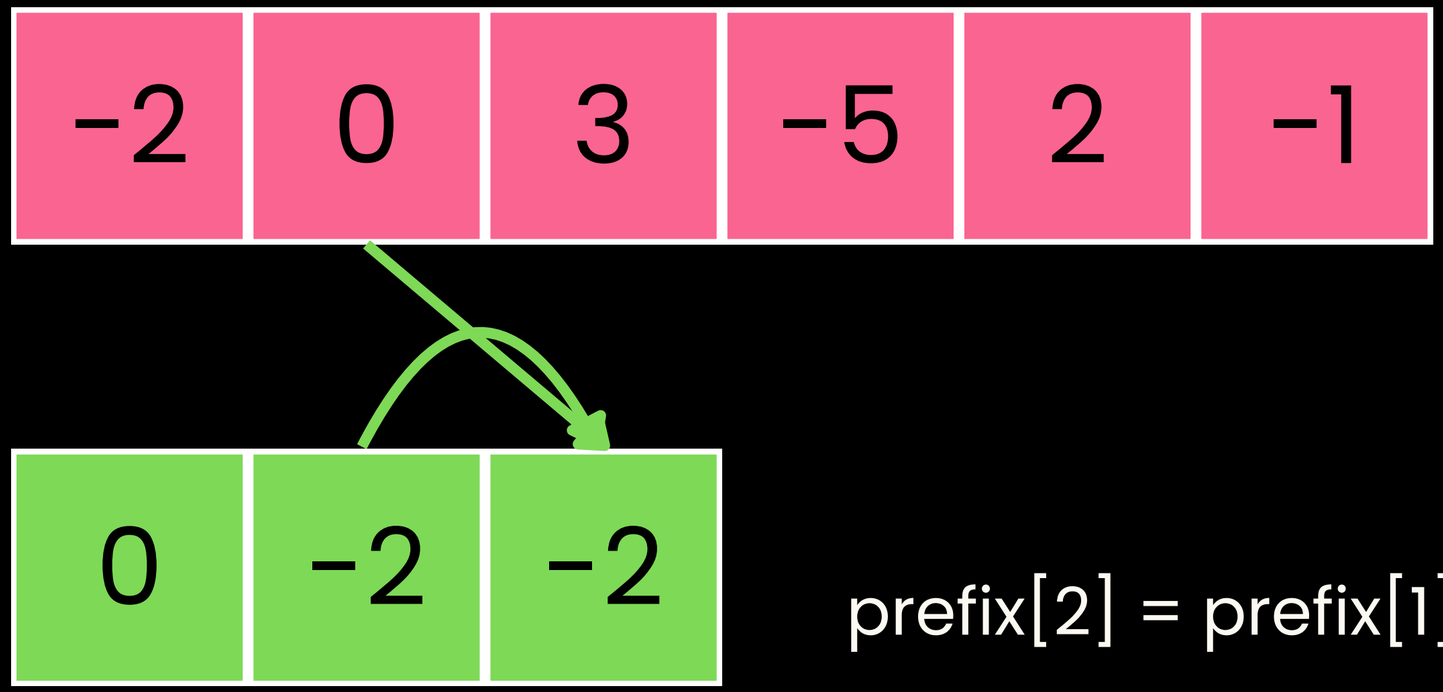
To find the next element, we can sum the -2 and the 0



prefix[1] = prefix[0] + array[0]

Solution

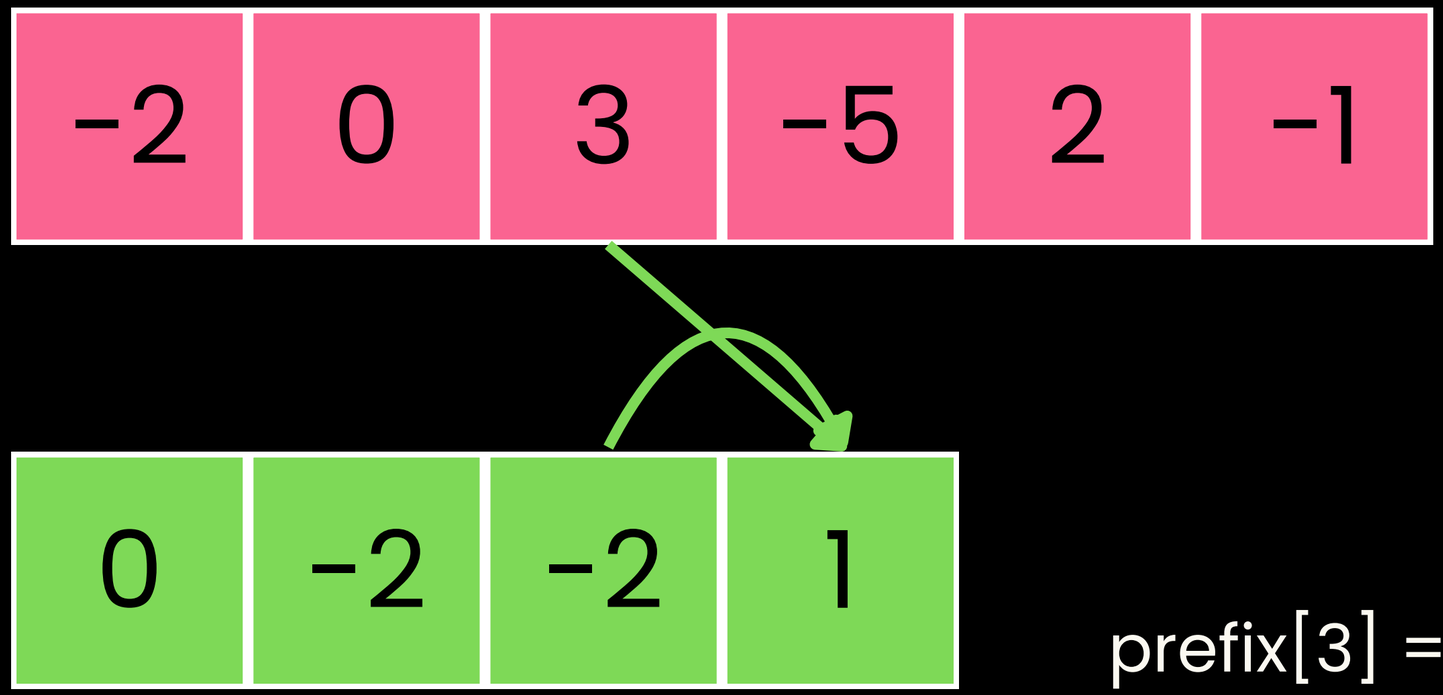
The next element can be found by summing the zero of the array and the -2 of the prefix sum



prefix[2] = prefix[1] + array[1]

Solution

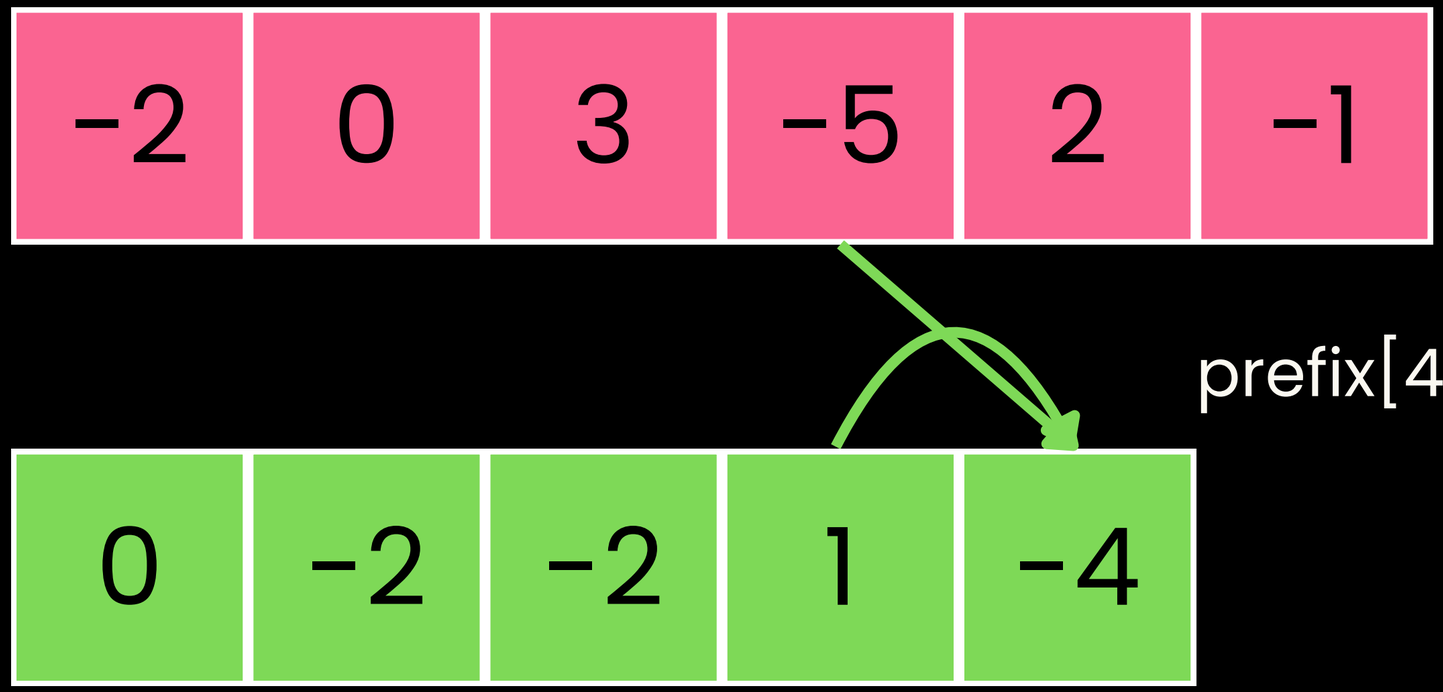
Continuing this, we can create every element of the prefix sum by summing the previous prefix sum and the next array element



prefix[3] = prefix[2] + array[2]

Solution

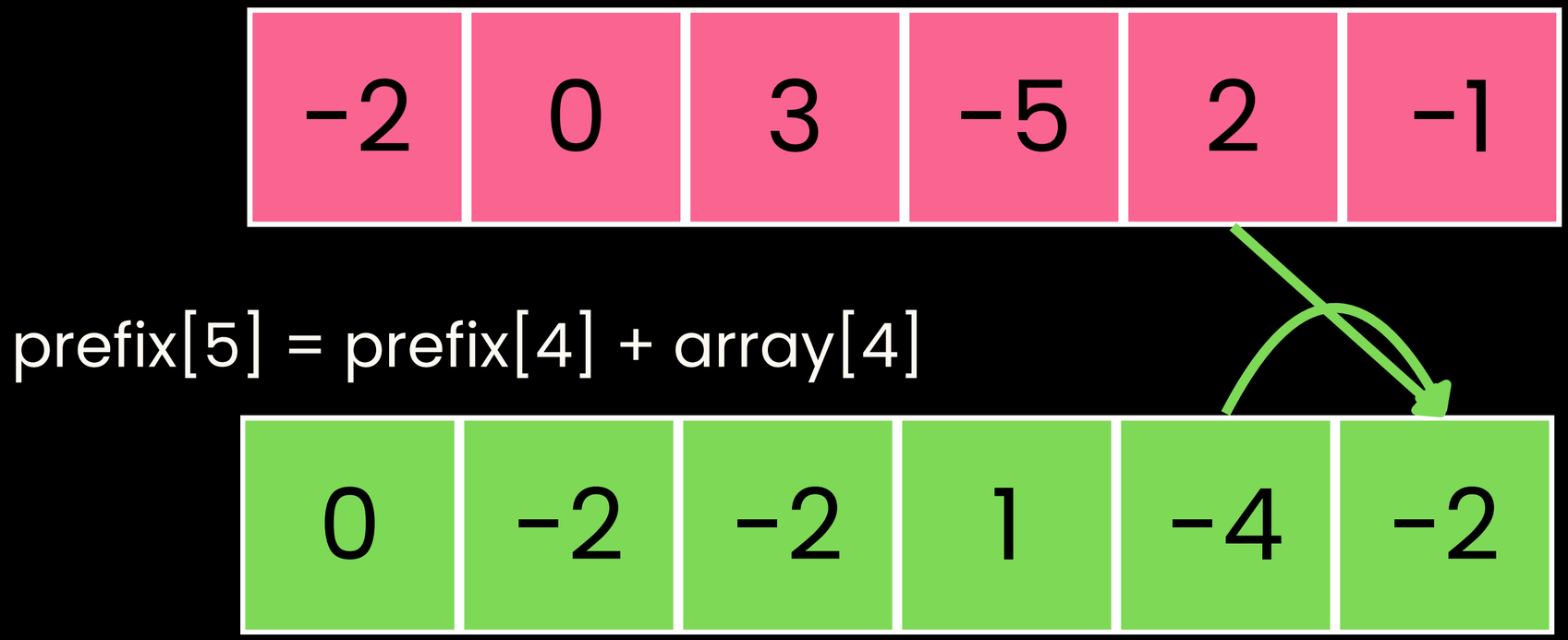
Continuing this, we can create every element of the prefix sum by summing the previous prefix sum and the next array element



prefix[4] = prefix[3] + array[3]

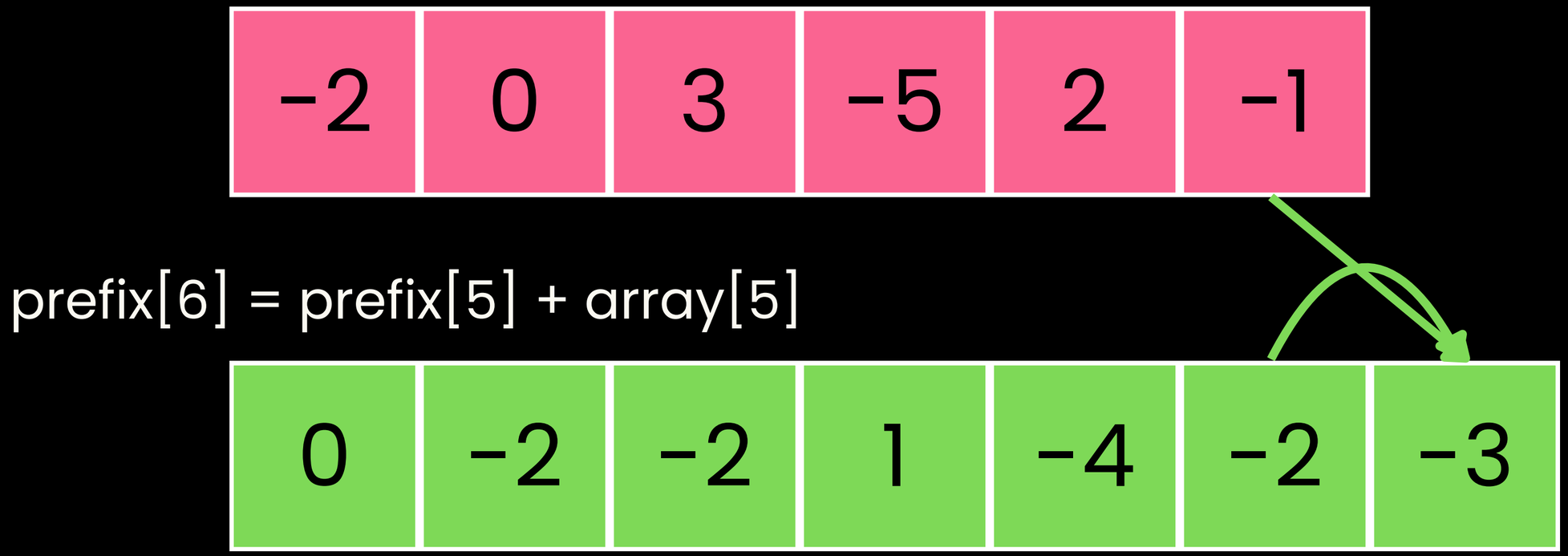
Solution

Continuing this, we can create every element of the prefix sum by summing the previous prefix sum and the next array element



Solution

Continuing this, we can create every element of the prefix sum by summing the previous prefix sum and the next array element



Prefix Sums Formulas:

To create the Prefix Sum Array:

prefix[0] = 0
prefix[i] = prefix[i-1] + array[i-1]

Array:	-2	0	3	-5	2	-1

Prefix:	0	-2	-2	1	-4	-2	-3

Querying the Prefix Sum

Ex. How can we find the sum from index 2 → 5

a = 2, b = 5

arr[2] + arr[3] + arr[4] + arr[5] = 3 - 5 + 2 - 1 = -1

Array:

-2	0	3	-5	2	-1

prefix[6] - prefix[2] = -3 - (-2) = -3 + 2 = -1

Prefix:

0	-2	-2	1	-4	-2	-3

Querying the Prefix Sum

Array:

-2	0	3	-5	2	-1

Prefix:

0	-2	-2	1	-4	-2	-3

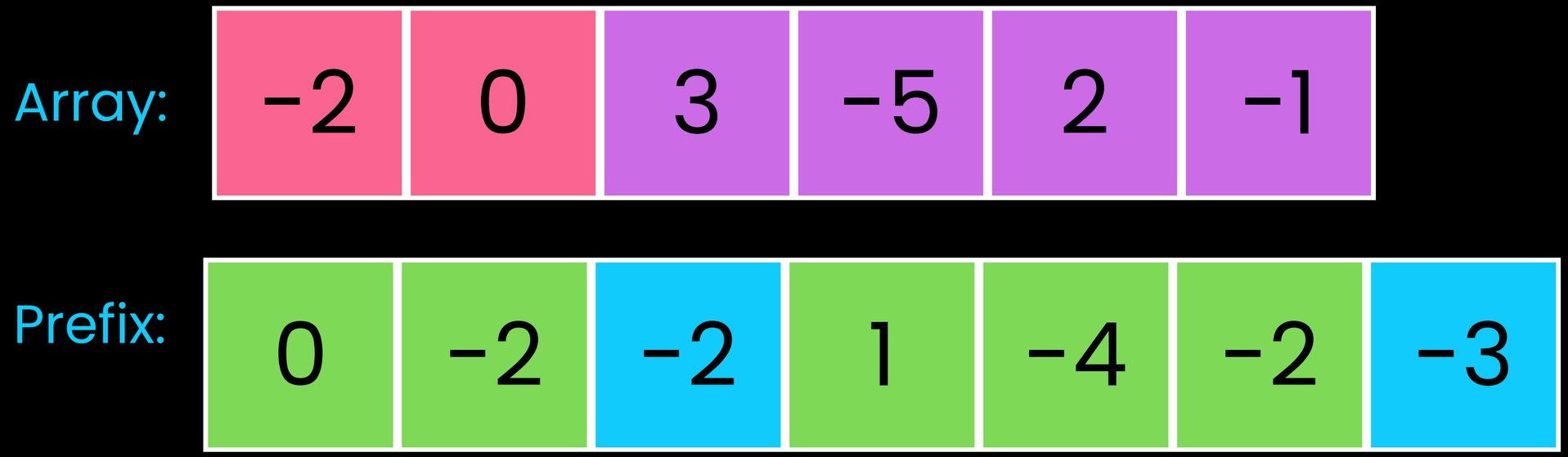
arr[2] + arr[3] + arr[4] + arr[5] = 3 - 5 + 2 - 1 = -1

prefix[6] - prefix[2] = -3 - (-2) = -3 + 2 = -1

Sum from 2 → 5 = prefix[6] - prefix[2]

How can we extrapolate this?

Querying the Prefix Sum



Sum from 2 → 5 = prefix[6] - prefix[2]

Sum from a → b = prefix[b+1] - prefix[a]

Java Solution

```java
class NumArray {
    private int[] prefix_sum;

    public NumArray(int[] nums) {
        // Initialize the prefix_sum array with a 0 at the start
        // (We allocate size + 1 to accommodate the leading 0)
        prefix_sum = new int[nums.length + 1];
        prefix_sum[0] = 0;

        // Compute the prefix sums
        for (int i = 0; i < nums.length; i++) {
            prefix_sum[i + 1] = prefix_sum[i] + nums[i];
        }
    }

    public int sumRange(int left, int right) {
        // Calculate the sum of elements from left to right using the prefix_sum array
        return prefix_sum[right + 1] - prefix_sum[left];
    }
}
```

Python Solution

```python
class NumArray:
    def __init__(self, nums: List[int]):
        # Initialize the prefix_sum array with a 0 at the start
        self.prefix_sum = [0]

        # Compute the prefix sums
        for num in nums:
            self.prefix_sum.append(self.prefix_sum[-1] + num)

    def sumRange(self, left: int, right: int) -> int:
        # Calculate the sum of elements from left to right using the prefix_sum array
        return self.prefix_sum[right + 1] - self.prefix_sum[left]

```

The Problem

https://leetcode.com/problems/contiguous-array/

525. Contiguous Array

Solved

Medium | Topics | Companies

Given a binary array `nums`, return *the maximum length of a contiguous subarray with an equal number of* `0` *and* `1`.

Example 1:

```
Input: nums = [0,1]
Output: 2
Explanation: [0, 1] is the longest contiguous subarray with an equal number of 0 and 1.
```

Example 2:

```
Input: nums = [0,1,0]
Output: 2
Explanation: [0, 1] (or [1, 0]) is a longest contiguous subarray with equal number of 0 and 1.
```

Example 3:

```
Input: nums = [0,1,1,1,1,1,0,0,0]
Output: 6
Explanation: [1,1,1,0,0,0] is the longest contiguous subarray with equal number of 0 and 1.
```

Constraints:

- `1 <= nums.length <= 10^5`
- `nums[i]` is either `0` or `1`.

Some Test Cases

525. Contiguous Array

Solved

Medium | Topics | Companies

Given a binary array `nums`, return *the maximum length of a contiguous subarray with an equal number of* `0` *and* `1`.

Example 1:

```
Input: nums = [0,1]
Output: 2
Explanation: [0, 1] is the longest contiguous subarray with an equal number of 0 and 1.
```

Example 2:

```
Input: nums = [0,1,0]
Output: 2
Explanation: [0, 1] (or [1, 0]) is a longest contiguous subarray with equal number of 0 and 1.
```

Example 3:

```
Input: nums = [0,1,1,1,1,1,0,0,0]
Output: 6
Explanation: [1,1,1,0,0,0] is the longest contiguous subarray with equal number of 0 and 1.
```

Constraints:

- `1 <= nums.length <= 10^5`
- `nums[i]` is either `0` or `1`.

Test case 2

nums = [0, 1, 0]

Ans = 2

nums = [0, 1, 0]

OR

nums = [0, 1, 0]

Test case 3

nums = [0, 1, 1, 1, 1, 1, 0, 0, 0]

Ans = 6

nums = [0, 1, 1, 1, 1, 1, 0, 0, 0]

Contiguous Array: Hint 1

Context: You can think of this problem of walking back and forth

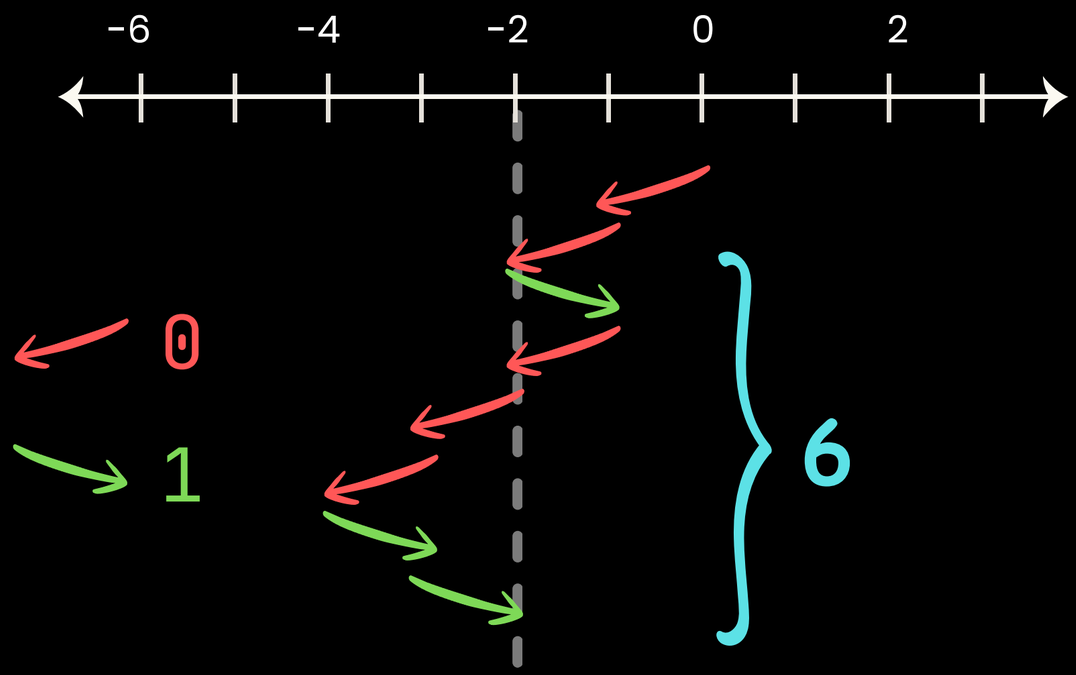
Specifically *0 is walk left and 1 is walk right.*



Contiguous Array: Solution

Sample test case:

`[0, 0, 1, 0, 0, 0, 1, 1]`



So, what can we take from this? Let's look at the problem again...

> Given a binary array `nums`, return *the maximum length of a contiguous subarray with an equal number of* `0` *and* `1`.

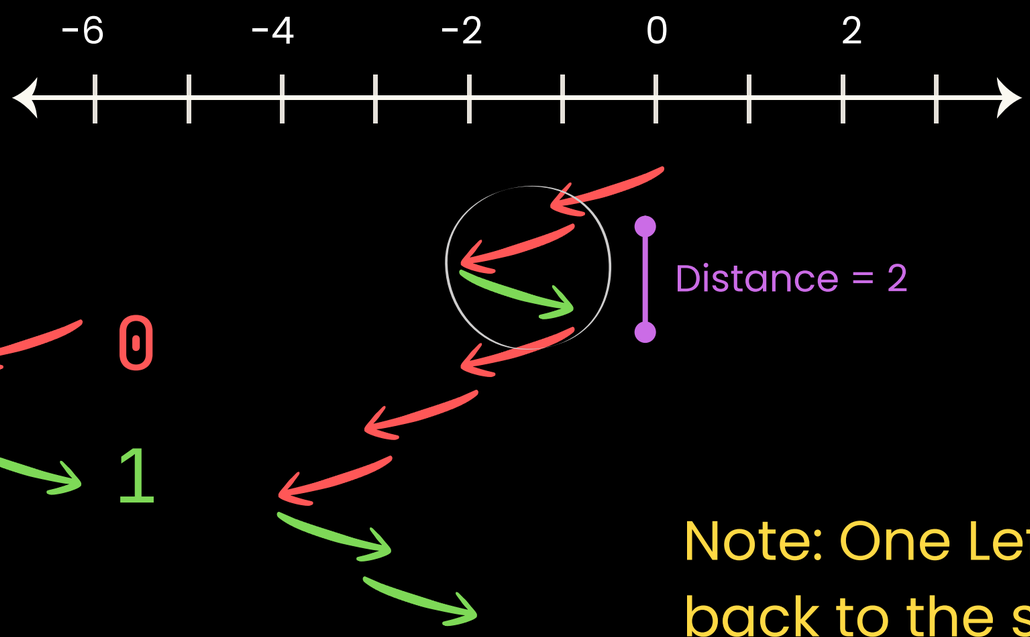
What does this mean for us?

Well, we need to determine **what characteristics are similar over all valid subarrays**

Examples of Valid Subarrays

Sample test case:

[0, 0, 1, 0, 0, 0, 1, 1]



Lets take a look at indices 1 to 2 (inclusive)

First, verify that there is an equal num of 1's and 0's → **One each**

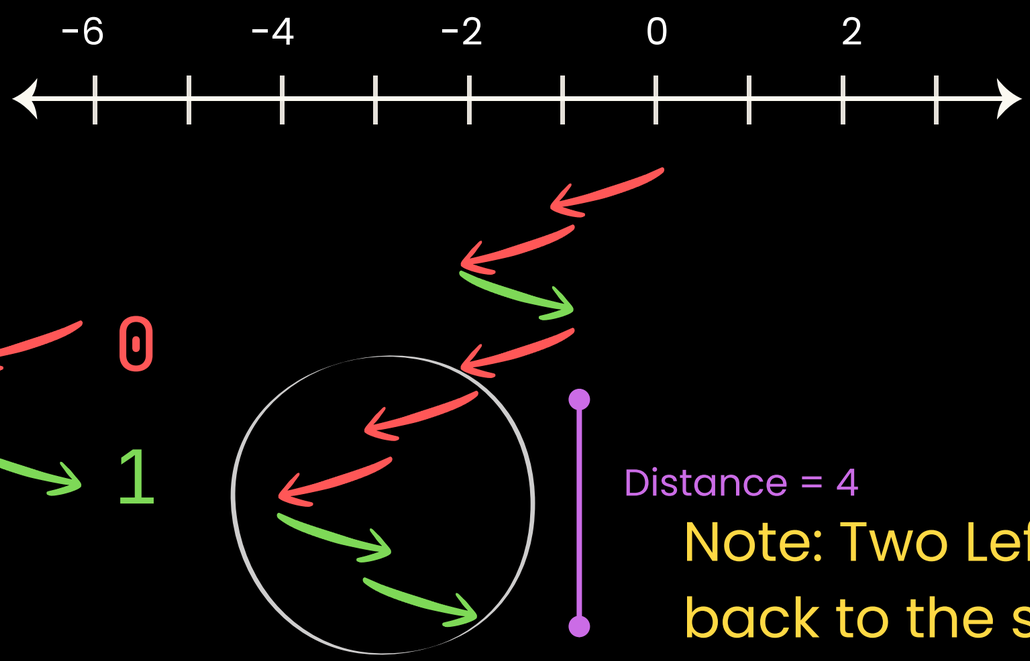
What does that look on the “visual”?

Note: One Left and One Right, goes back to the same **x-position (x = -1)**

Examples of Valid Subarrays

Sample test case:

[0, 0, 1, 0, 0, 0, 1, 1]



What about indices 4 to 7 inclusive?

Remeber, verify that there is an equal num of 1's and 0's → **Two each**

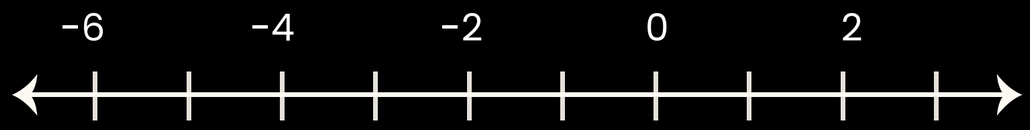
What does that look on the “visual”?

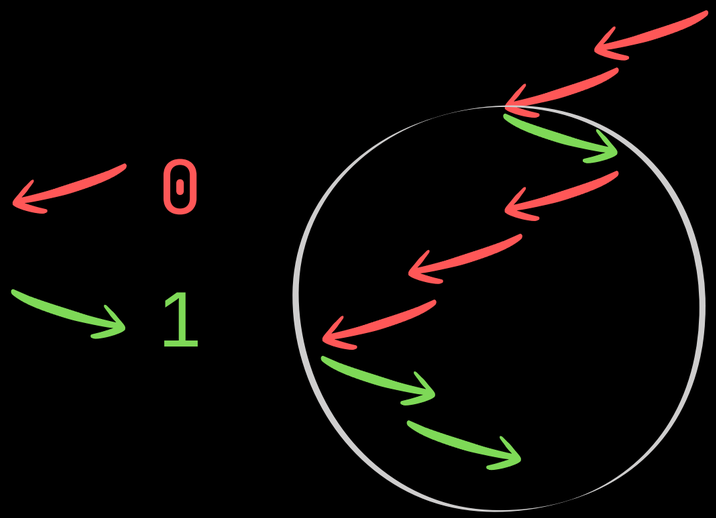
Note: Two Left and Two Right, goes back to the same **x-position (x = -2)**

Final Example of Valid Subarrays

Sample test case:

[0, 0, 1, 0, 0, 0, 1, 1]





What about indices 2 to 7 inclusive?

Remeber, verify that there is an equal num of 1's and 0's → **Three each**

What does that look on the “visual”?

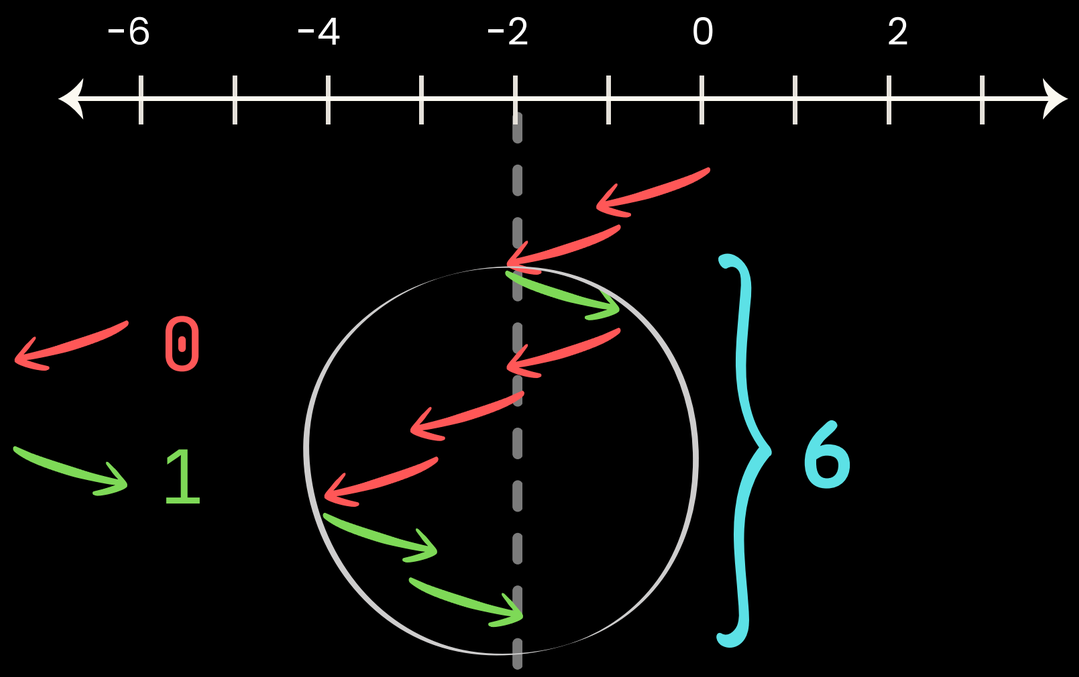
Note: Three Left and Three Right, goes back to the same **x-position (x = -2)**

THE ORDER OF LEFT AND RIGHTS ARE IRRELEVANT, because 3 left and 3 right = no displacement

Building on the intuition

Sample test case:

`[0, 0, 1, 0, 0, 0, 1, 1]`



Basically we need to find times when we reach a place we've seen before

We need to both store

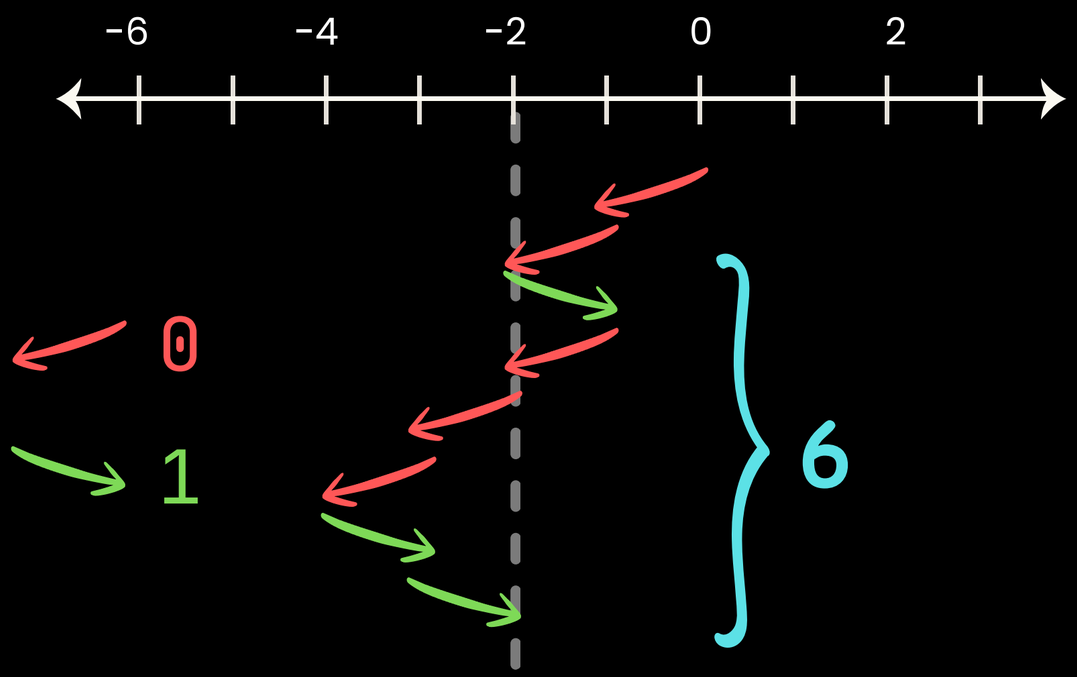
1. where we are currently (in a running sum)

2. Where we've been before (in a hashmap/dictionary)

Building on the intuition - final step

Sample test case:

[0, 0, 1, 0, 0, 0, 1, 1]



1. where we are currently (in a running sum)

2. Where we've been before (in a hashmap/dictionary)

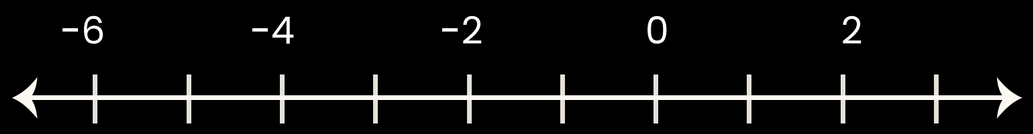
With this info, our BEST/MAXIMUM subarray is based on the FIRST and LAST time we've reached a value

We will use the dictionary to help us remember "that first time"

Seeing it in action

Sample test case:

[0, 0, 1, 0, 0, 0, 1, 1]



0

1

maxAnswer = 0

runningSum = 0

lookupDict =

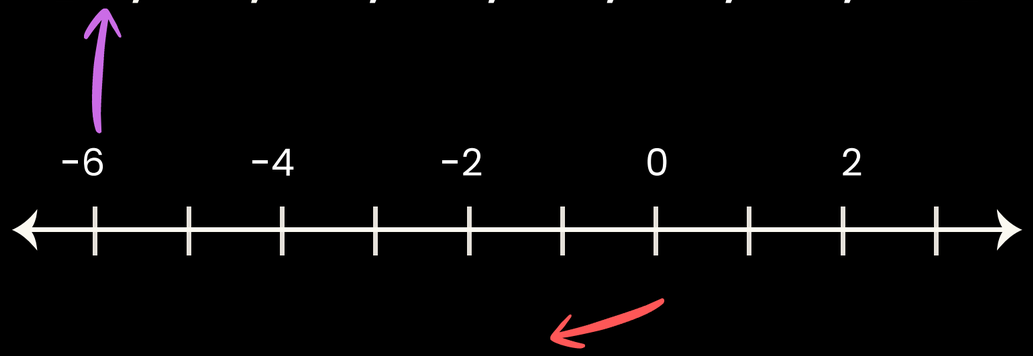
```
{
    0: -1
}
```

We start by saying that the FIRST time we've seen a runningSum = 0 was before index = -1 **(before we even start iterating through the array!)**

Seeing it in action

Sample test case:

[0, 0, 1, 0, 0, 0, 1, 1]



0

1

Have we seen
RunningSum = -1?

maxAnswer = 0

runningSum = -1

lookupDict =
{
0: -1
-1: 0
}

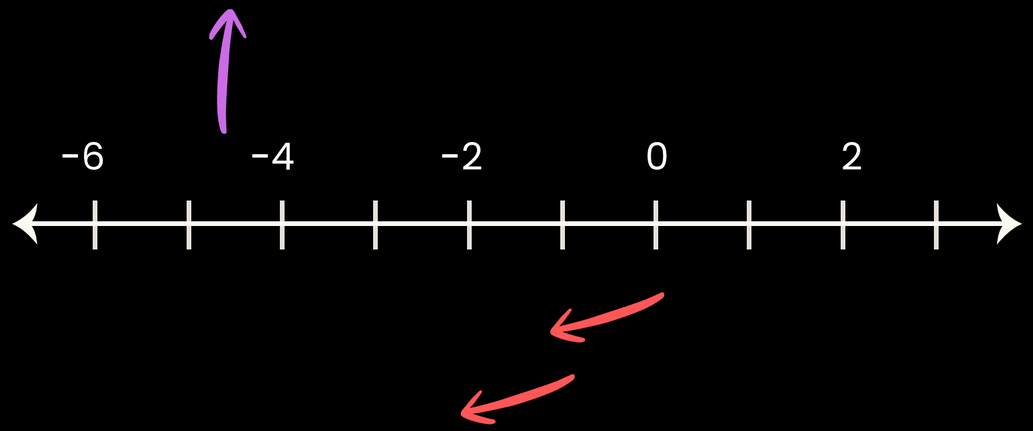
0 means go left (-1)
Running Sum is now -1

Nope, so store it in dict
RunningSum = -1 with
index = 0

Seeing it in action

Sample test case:

[0, 0, 1, 0, 0, 0, 1, 1]



0

1

Have we seen RunningSum = -2?

maxAnswer = 0

runningSum = -2

lookupDict =

```
{
    0: -1
    -1: 0
    -2: 1
}
```

0 means go left (-1)

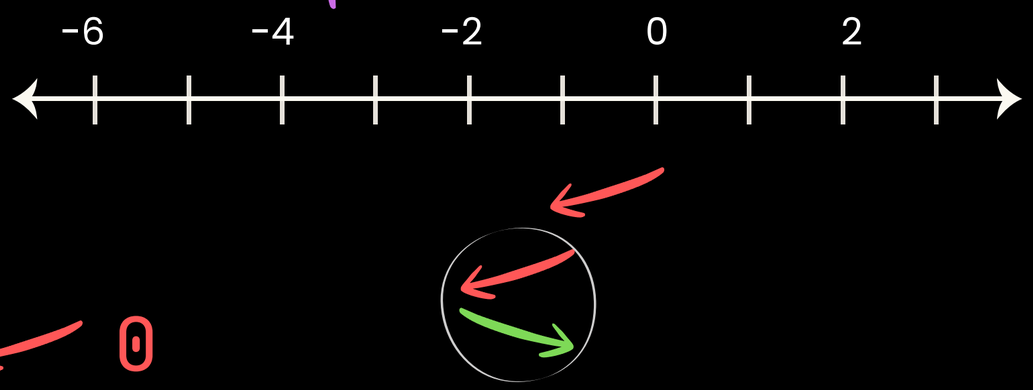
Running Sum is now -2

Nope, so store it in dict RunningSum = -2 with index = 1

Seeing it in action

Sample test case:

[0, 0, 1, 0, 0, 0, 1, 1]



0

1

Have we seen RunningSum = -1?

maxAnswer = **2**

runningSum = -1

lookupDict =

{

0: -1

-1: 0

-2: 1

}

USE THIS

1 means go right (+1)

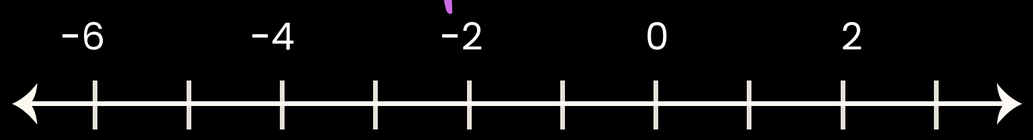
Running Sum is now -1

Yep, so maxAnswer = currIndex (2) - firstSeen (0) = **2**

Seeing it in action

Sample test case:

[0, 0, 1, 0, 0, 0, 1, 1]



0

1

Have we seen RunningSum = -2?

maxAnswer = 2

runningSum = -2

lookupDict =

{

0: -1

-1: 0

-2: 1

}

USE THIS

0 means go left (-1)

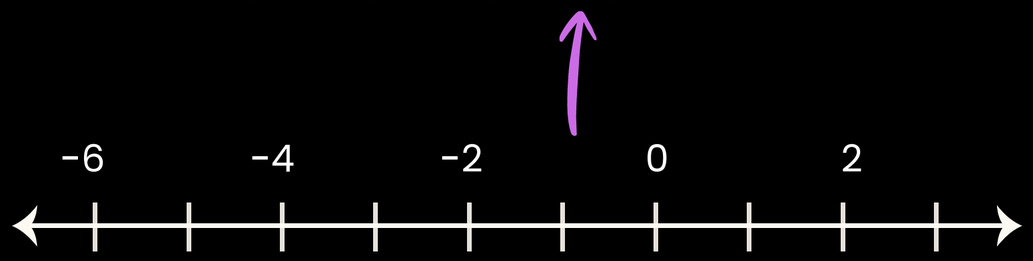
Running Sum is now -2

Yep, so maxAnswer = currIndex (3) - firstSeen (1) = 2

Seeing it in action

Sample test case:

[0, 0, 1, 0, 0, 0, 1, 1]



0

1

Have we seen RunningSum = -3?

Nope, so store it in dict

RunningSum = -3 seen before

index = 4

maxAnswer = 2

runningSum = -3

lookupDict =

```
{
    0: -1
    -1: 0
    -2: 1
    -3: 4
}
```

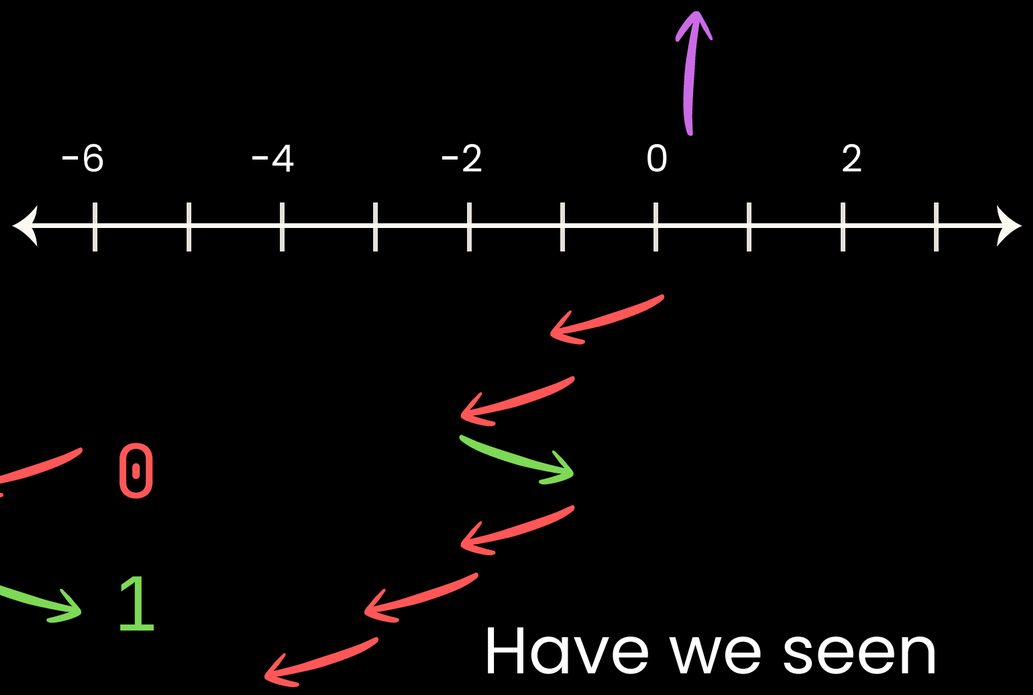
0 means go left (-1)

Running Sum is now -3

Seeing it in action

Sample test case:

[0, 0, 1, 0, 0, 0, 1, 1]



Have we seen RunningSum = -4?

maxAnswer = 2

runningSum = -4

lookupDict =

```
{
    0: -1
    -1: 0
    -2: 1
    -3: 4
    -4: 5
}
```

0 means go left (-1)

Running Sum is now -4

Nope, so store it in dict

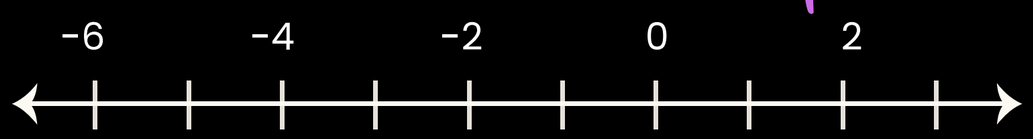
RunningSum = -4 seen before

index = 5

Seeing it in action

Sample test case:

[0, 0, 1, 0, 0, 0, 1, 1]



0

1



Have we seen RunningSum = -3?

maxAnswer = 2

runningSum = -3

lookupDict =

{

0: -1

-1: 0

-2: 1

-3: 4

-4: 5

}

USE THIS

1 means go right (+1)

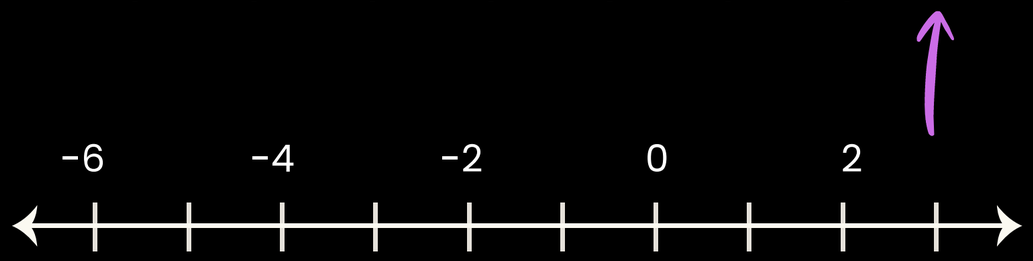
Running Sum is now -3

Yep, so maxAnswer = curIndex (6) - firstSeen (4) = 2

Seeing it in action

Sample test case:

[0, 0, 1, 0, 0, 0, 1, 1]



0

1

Have we seen RunningSum = -2?

maxAnswer = **6**

runningSum = -2

lookupDict =

```
{
    0: -1
    -1: 0
    -2: 1
    -3: 4
    -4: 5
}
```

USE THIS

1 means go right (+1)

Running Sum is now -2

Yep, so maxAnswer = currIndex (7) - firstSeen (1) = **6**

Contiguous Array: Java's O(n) solution

Implementation Note:

We need to set -1 as our value for displacement x:0, because we are there before we even start moving/ iterating through nums.

Code

Java Auto

```java
class Solution {
    public int findMaxLength(int[] nums) {
        // Initializes a HashMap
        HashMap<Integer, Integer> lookupHashmap =
            new HashMap<Integer, Integer>();

        // Base case: Signifies sum of zero at start of the list (index -1)
        lookupHashmap.put(0, -1);
        int maxLen = 0;
        int currentSum = 0;

        // Iterate through array
        for (int i = 0; i < nums.length; i++) {
            // Update currentSum: treat 0 as -1, 1 as 1
            currentSum += (nums[i] == 1) ? 1 : -1;

            if (lookupHashmap.getOrDefault(currentSum, -2) != -2) {
                // We have seen this sum before
                int length = i - lookupHashmap.get(currentSum);
                if (length > maxLen) {
                    maxLen = length;
                }
            } else {
                // First time seeing this sum, remember the index
                lookupHashmap.put(currentSum, i);
            }
        }

        return maxLen;
    }
}
```

Contiguous Array: Python's O(n) solution

Implementation Note:

We need to set -1 as our value for displacement x:0, because we are there before we even start moving/ iterating through nums.

Code

Python3 Auto

```python
class Solution:
    def findMaxLength(self, nums: List[int]) -> int:
        # Initializes a dictionary
        lookup_dict = {}

        # Base case: Signifies sum of zero at start of the list (index -1)
        lookup_dict[0] = -1
        max_len = 0
        current_sum = 0

        for i, num in enumerate(nums):
            # Update current_sum: treat 0 as -1, 1 as 1
            current_sum += 1 if num == 1 else -1

            if lookup_dict.get(current_sum, -2) != -2: # We have seen this sum before
                length = i - lookup_dict[array_index]
                if length > max_len:
                    max_len = length

            else: # First time seeing this sum
                lookup_dict[array_index] = i

        return max_len
```

Hacker Devils

Presented by Sohum Sheth and Dean Richardson

THANK YOU

for your time and attention

