

## Grokking the Coding Interview: Patterns for Coding Questions

1% completed

 Search Course

### Introduction

- Who should take this course?
- Course Overview

### Pattern: Sliding Window

- Introduction
- Maximum Sum Subarray of Size K (easy)
- Smallest Subarray with a given sum (easy)
- Longest Substring with K Distinct Characters (medium)
- Fruits into Baskets (medium)
- No-repeat Substring (hard)
- Longest Substring with Same Letters after Replacement (hard)
- Longest Subarray with Ones after Replacement (hard)
- Problem Challenge 1
- Solution Review: Problem Challenge 1
- Problem Challenge 2
- Solution Review: Problem Challenge 2
- Problem Challenge 3
- Solution Review: Problem Challenge 3
- Problem Challenge 4
- Solution Review: Problem Challenge 4

### Pattern: Two Pointers

- Introduction
- Pair with Target Sum (easy)
- Remove Duplicates (easy)
- Squaring a Sorted Array (easy)
- Triplet Sum to Zero (medium)
- Triplet Sum Close to Target (medium)
- Triplets with Smaller Sum (medium)
- Subarrays with Product Less than a Target (medium)
- Dutch National Flag Problem (medium)
- Problem Challenge 1
- Solution Review: Problem Challenge 1
- Problem Challenge 2
- Solution Review: Problem Challenge 2
- Problem Challenge 3
- Solution Review: Problem Challenge 3

### Pattern: Fast & Slow pointers

- Introduction
- LinkedList Cycle (easy)
- Start of LinkedList Cycle (medium)

## Introduction

In many problems dealing with an array (or a LinkedList), we are asked to find or calculate something among all the contiguous subarrays (or sublists) of a given size. For example, take a look at this problem:

Given an array, find the average of all contiguous subarrays of size 'K' in it.

Let's understand this problem with a real input:

Array: [1, 3, 2, 6, -1, 4, 1, 8, 2], K=5

Here, we are asked to find the average of all contiguous subarrays of size '5' in the given array. Let's solve this:

- For the first 5 numbers (subarray from index 0-4), the average is:  $(1 + 3 + 2 + 6 - 1)/5 \Rightarrow 2.2$
- The average of next 5 numbers (subarray from index 1-5) is:  $(3 + 2 + 6 - 1 + 4)/5 \Rightarrow 2.8$
- For the next 5 numbers (subarray from index 2-6), the average is:  $(2 + 6 - 1 + 4 + 1)/5 \Rightarrow 2.4$
- ...

Here is the final output containing the averages of all contiguous subarrays of size 5:

Output: [2.2, 2.8, 2.4, 3.6, 2.8]

A brute-force algorithm will be to calculate the sum of every 5-element contiguous subarray of the given array and divide the sum by '5' to find the average. This is what the algorithm will look like:

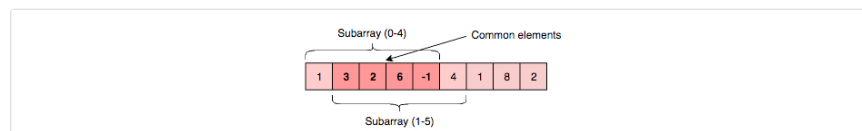
```

1 import java.util.Arrays;
2
3 class AverageOfSubarrayOfSizeK {
4     public static double[] findAverages(int K, int[] arr) {
5         double[] result = new double[arr.length - K + 1];
6         for (int i = 0; i <= arr.length - K; i++) {
7             // find sum of next 'K' elements
8             double sum = 0;
9             for (int j = i; j < i + K; j++)
10                 sum += arr[j];
11             result[i] = sum / K; // calculate average
12         }
13
14         return result;
15     }
16
17     public static void main(String[] args) {
18         double[] result = AverageOfSubarrayOfSizeK.findAverages(5, new int[] { 1, 3, 2, 6, -1, 4, 1, 8, 2 });
19         System.out.println("Averages of subarrays of size K: " + Arrays.toString(result));
20     }
21 }
  
```

**Time complexity:** Since for every element of the input array, we are calculating the sum of its next 'K' elements, the time complexity of the above algorithm will be  $O(N * K)$  where 'N' is the number of elements in the input array.

Can we find a better solution? Do you see any inefficiency in the above approach?

The inefficiency is that for any two consecutive subarrays of size '5', the overlapping part (which will contain four elements) will be evaluated twice. For example, take the above-mentioned input:



As you can see, there are four overlapping elements between the subarray (indexed from 0-4) and the subarray (indexed from 1-5). Can we somehow reuse the `sum` we have calculated for the overlapping elements?

The efficient way to solve this problem would be to visualize each contiguous subarray as a sliding window of '5' elements. This means that when we move on to the next subarray, we will slide the window by one element. So, to reuse the `sum` from the previous subarray, we will subtract the element going out of the window and add the element now being included in the sliding window. This will save us from going through the whole subarray to find the `sum` and, as a result, the algorithm complexity will reduce to  $O(N)$ .

Sliding window -->

Happy Number (medium)

Middle of the LinkedList (easy)

Problem Challenge 1

Solution Review: Problem Challenge 1

Problem Challenge 2

Solution Review: Problem Challenge 2

Problem Challenge 3

Solution Review: Problem Challenge 3

Pattern: Merge Intervals

Introduction

Merge Intervals (medium)

Insert Interval (medium)

Intervals Intersection (medium)

Conflicting Appointments (medium)

Problem Challenge 1

Solution Review: Problem Challenge 1

Problem Challenge 2

Solution Review: Problem Challenge 2

Problem Challenge 3

Solution Review: Problem Challenge 3

Pattern: Cyclic Sort

Introduction

Cyclic Sort (easy)

Find the Missing Number (easy)

Find all Missing Numbers (easy)

Find the Duplicate Number (easy)

Find all Duplicate Numbers (easy)

Problem Challenge 1

Solution Review: Problem Challenge 1

Problem Challenge 2

Solution Review: Problem Challenge 2

Problem Challenge 3

Solution Review: Problem Challenge 3

Pattern: In-place Reversal of a LinkedList

Introduction

MW

Explore

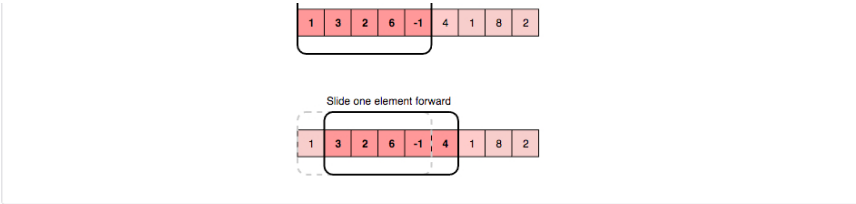
Tracks

My Courses

Edpresso

Refer a Friend

Create



Here is the algorithm for the **Sliding Window** approach:

Java Python3 C++ JS

```
1 import java.util.Arrays;
2
3 class AverageOfSubarrayOfSizeK {
4     public static double[] findAverages(int K, int[] arr) {
5         double[] result = new double[arr.length - K + 1];
6         double windowSum = 0;
7         int windowStart = 0;
8         for (int windowEnd = 0; windowEnd < arr.length; windowEnd++) {
9             windowSum += arr[windowEnd]; // add the next element
10            // slide the window, we don't need to slide if we've not hit the required window size of 'k'
11            if (windowEnd >= K - 1) {
12                result[windowStart] = windowSum / K; // calculate the average
13                windowSum -= arr[windowStart]; // subtract the element going out
14                windowStart++; // slide the window ahead
15            }
16        }
17        return result;
18    }
19
20    public static void main(String[] args) {
21        double[] result = AverageOfSubarrayOfSizeK.findAverages(5, new int[] { 1, 3, 2, 6, -1, 4, 1, 8, 2 });
22        System.out.println("Averages of subarrays of size K: " + Arrays.toString(result));
23    }
24 }
25 }
```

RUN SAVE RESET ↺

In the following chapters, we will apply the **Sliding Window** approach to solve a few problems.

In some problems, the size of the sliding window is not fixed. We have to expand or shrink the window based on the problem constraints. We will see a few examples of such problems in the next chapters.

Let's jump onto our first problem and apply the **Sliding Window** pattern.

← Back

Course Overview

✓ MARK AS COMPLETED

Next →

Maximum Sum Subarray of Size K (ea...

🔊 Report an Issue