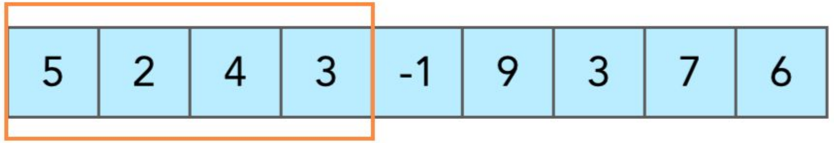
Sliding Window



Sliding window —> —>

Lecture Flow

- Pre-requisites
- Introduction to problem-solving technique
- Basic Concepts
- One Dimensional problems
- Two Dimensional problems
- Variants
- Pitfalls
- Complexity Analysis
- Applications of sliding window
- Practice Questions

Pre-requisites

1. Arrays
2. Dictionary/Maps
3. Sets/Hash Sets
4. Two Pointers

Introduction

- The **sliding window technique** is a problem-solving technique used to solve various problems involving arrays or strings.

- Unlocks solutions for finding max/min values, counting elements, and pattern detection

- Dynamic window glides over input, achieving linear time complexity

Basic Concepts

The sliding window technique consists of two key components: **window size** and **window movement**.

- **window size** - the number of elements in the array or string that the window encompasses.

- **window movement** - the number of elements by which the window moves after each step.

One Dimensional problems

One-dimensional problems can be solved using the sliding window technique by creating a window of fixed size that moves over the array, one element at a time.

One Dimensional problems Cont.

Consider the problem of finding the maximum sum of k consecutive elements in an array.

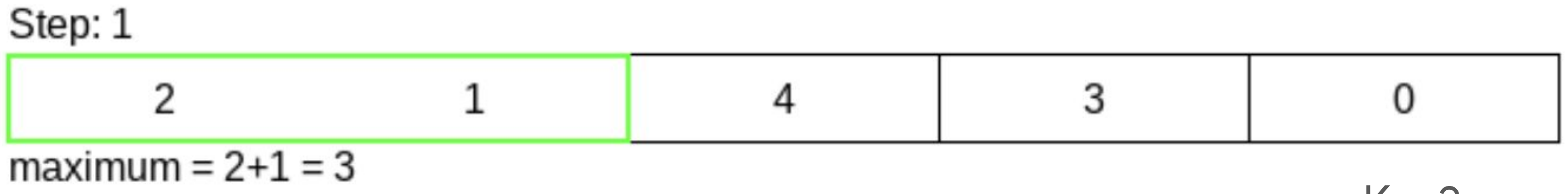
How can we use sliding window to solve this problem ?

One Dimensional Sliding Window Simulation

2	1	4	3	0

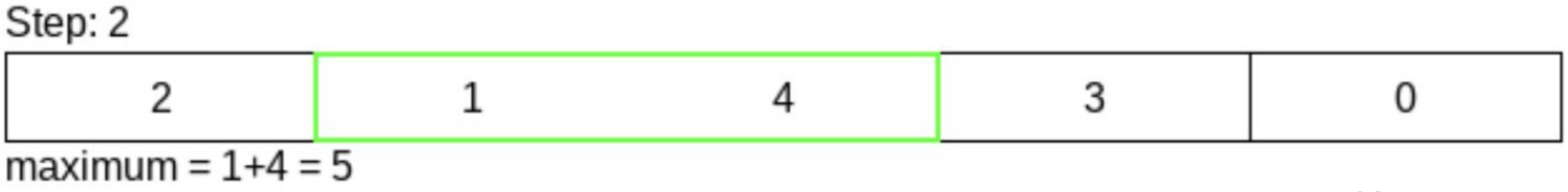
K = 2

One Dimensional Sliding Window Simulation



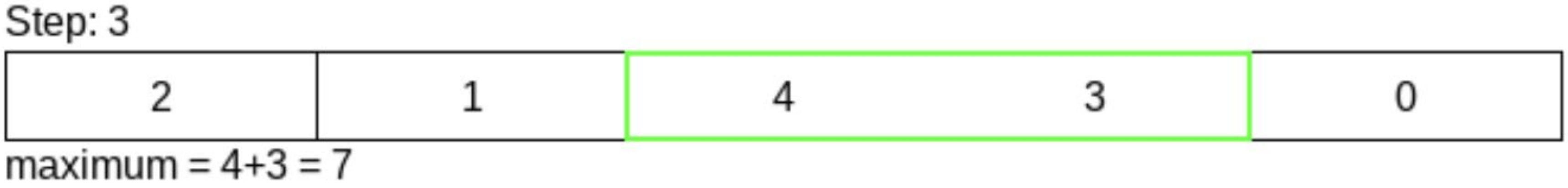
K = 2

One Dimensional Sliding Window Simulation



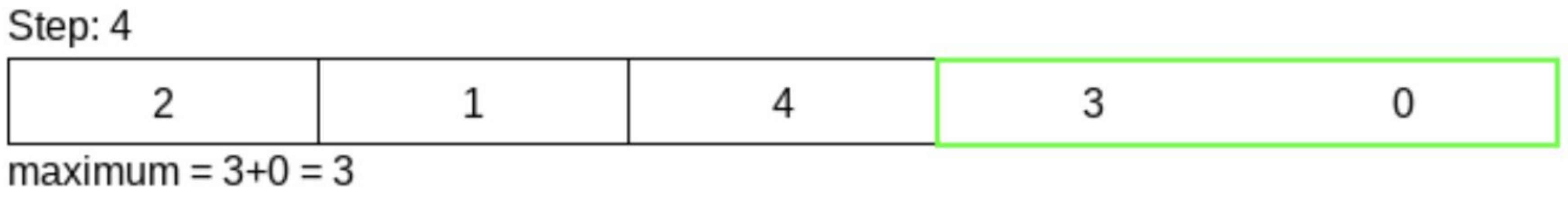
K = 2

One Dimensional Sliding Window Simulation



K = 2

One Dimensional Sliding Window Simulation



Maximum sum of two items from this array is 7 and the pair is {4,3}.

K = 2

Two Dimensional problems

Two-dimensional problems can be solved using the sliding window technique by creating a window of fixed size that moves over the matrix or array, one row or column at a time.

Two Dimensional problems Cont.

Consider the problem of finding the maximum sum of a submatrix of size k x k in a matrix.

How can we solve this problem using sliding window ?

Two Dimensional problems Cont.

To solve this problem using the sliding window technique,

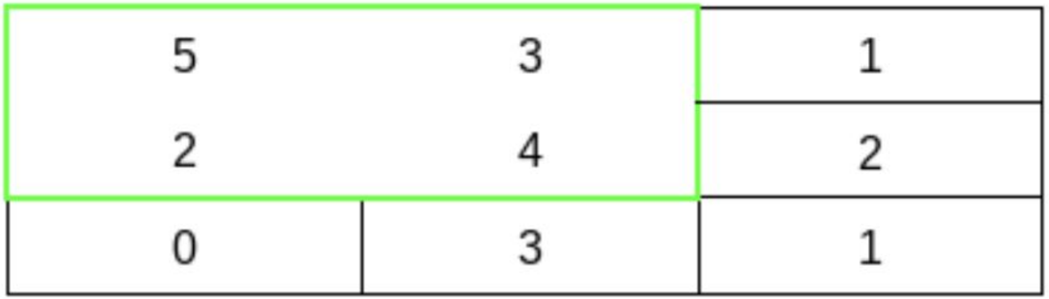
- We would create a window of size k x k and
- Move it over the matrix, summing up the elements in each window and
- Keeping track of the maximum sum we have seen so far.

Two Dimensional Sliding Window Simulation

5	3	1
2	4	2
0	3	1

Two Dimensional Sliding Window Simulation

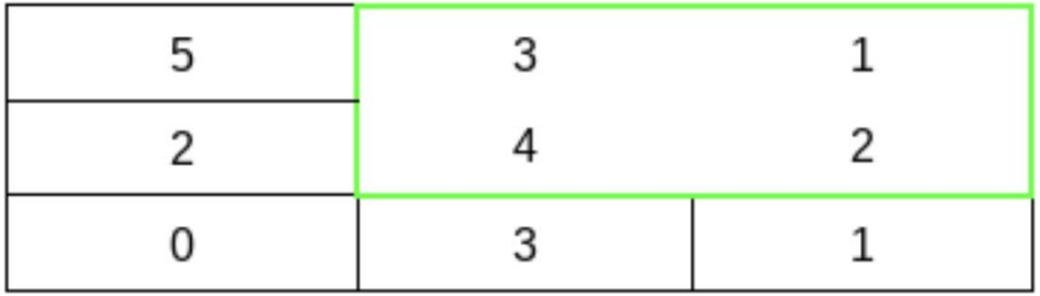
Step: 1



maximum = 5+3+2+4 = 14

Two Dimensional Sliding Window Simulation

Step: 2

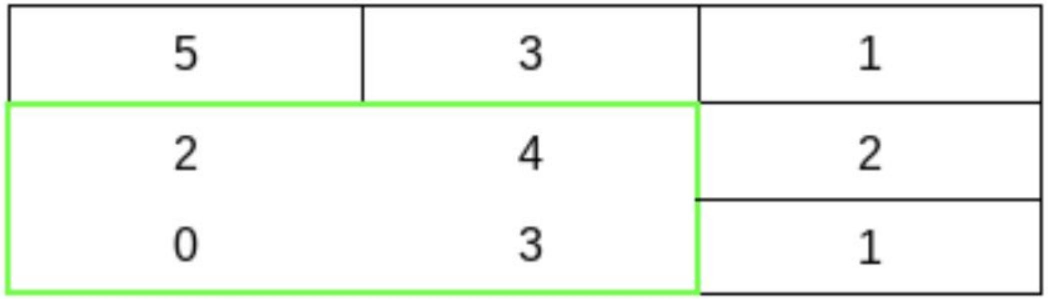


5	3	1
2	4	2
0	3	1

maximum = 3+1+4+2 = 10

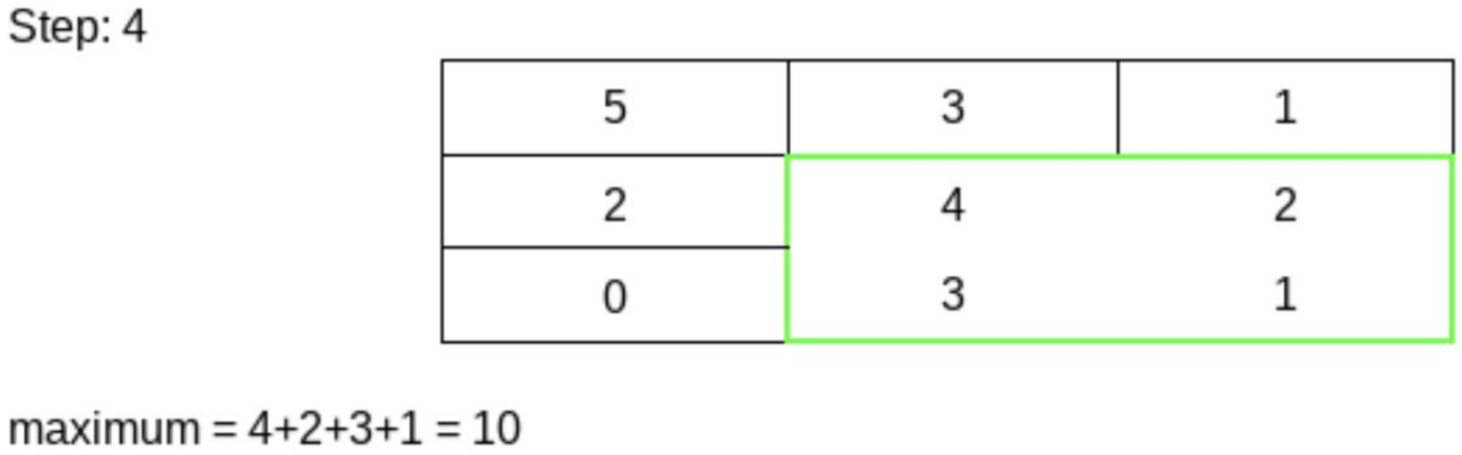
Two Dimensional Sliding Window Simulation

Step: 3



maximum = 2+4+0+3 = 9

Two Dimensional Sliding Window Simulation



Step: 4

5	3	1
2	4	2
0	3	1

maximum = 4+2+3+1 = 10

Variants

There are several types of sliding window:

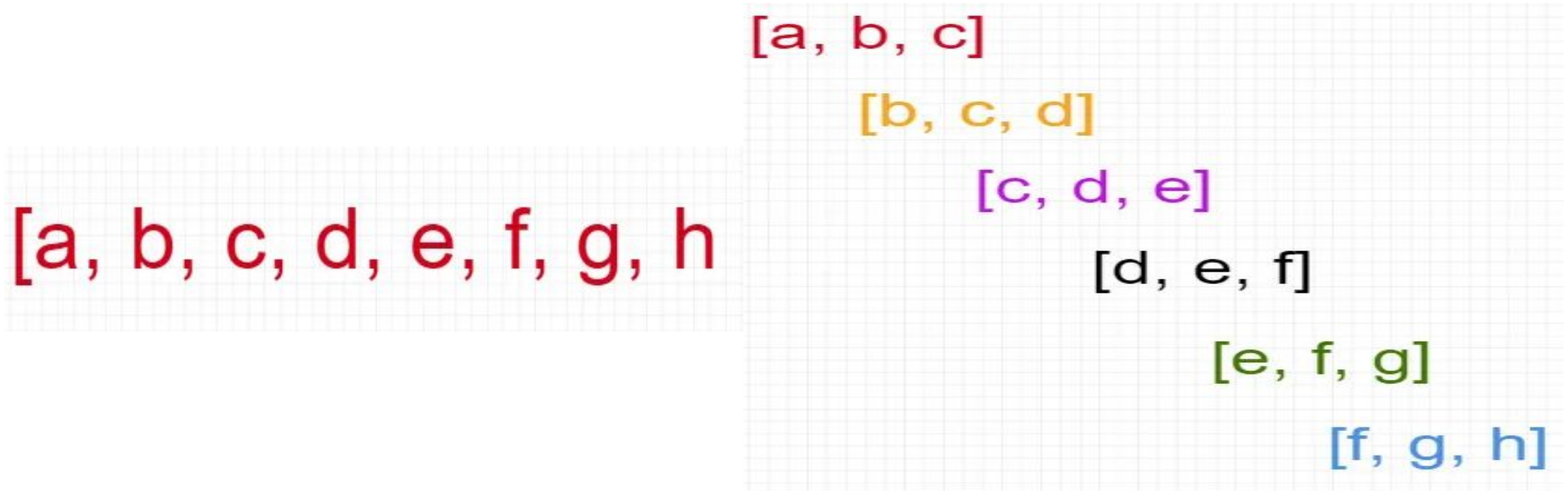
1. Fixed Window Length k
   i. Optimization (max, min)
   ii. Counting
2. Dynamic Sliding Window

Variant I

Fixed Size Sliding Window: Optimization

In this technique, a window of fixed size is moved over the input data, and some operation is performed on the data within the window.

For k = 3:



Fixed Size Sliding Window: Optimization

Example: Given an array of integers, find the maximum sum of any contiguous subarray of size 3.

Fixed Size Sliding Window: Optimization

```
max_sum = 0
```

```
[3,1,4,2,3,0]
```

Fixed Size Sliding Window: Optimization

`max_sum = 8`

$[\underline{3,1,4},2,3,0]$

Fixed Size Sliding Window: Optimization

max_sum = 8

[3, 1, 4, 2, 3, 0]

Fixed Size Sliding Window: Optimization

`max_sum = 9`

$$[3,1,\underline{4,2,3},0]$$

Fixed Size Sliding Window: Optimization

```
max_sum = 9
```

```
[3,1,4,2,3,0]
```

Pair Programming

Problem Link

Time and space complexity?

Fixed-size sliding window

- Time complexity: **O(n)**
- Space complexity: **O(1)**

Variant II

Fixed Size Sliding Window: Counting

In this technique, a sliding window is used to count the number of occurrences of a particular item in the input data.

For example, given two strings s and p, find the number of anagrams of p in s.

Fixed Size Sliding Window: Counting

```
s = "cbadabac"
p = "abc"
```

Fixed Size Sliding Window: Counting

```
anagrams = 0
```

```
s = "cbadabac"
```

```
target = { a:1, b:1, c:1 }   // represent p as a counter
```

Fixed Size Sliding Window: Counting

```
anagrams = 1
```

```
s = "cbadabac"
```

```
target = { a:1, b:1, c:1 }
window = { a:1, b:1, c:1 }   // represent each window as a
counter
                             and compare to target
```

How did we determine the size of the window?

Fixed Size Sliding Window: Counting

```
anagrams = 1
```

```
s = “cbadabac”
```

```
target = { a:1, b:1, c:1 }
window = { a:1, b:1, d:1 }
```

Africa To Silicon Valley

Fixed Size Sliding Window: Counting

```
anagrams = 1
```

```
s = “cbadabac”
```

```
target = { a:1, b:1, c:1 }
window = { a:2, d:1 }
```

Fixed Size Sliding Window: Counting

```
anagrams = 1
```

```
s = “cbadabac”
```

```
target = { a:1, b:1, c:1 }
window = { a:1, b:1, d:1 }
```

A2SV
Africa To Silicon Valley

Fixed Size Sliding Window: Counting

```
anagrams = 1
```

s = "cbad<u>aba</u>c"

```
target = { a:1, b:1, c:1 }

window = { a:2, b:1 }
```

Fixed Size Sliding Window: Counting

```
anagrams = 2
```

```
s = "cbadabac"
```

```
target = { a:1, b:1, c:1 }
window = { a:1, b:1, c:1 }
```

Pair Programming

[Problem Link]

Time and space complexity?

Fixed-size sliding window

- Time complexity: **O(n)**
- Space complexity: **O(1)**

Variant III

Dynamic Sliding Window

In this technique, the size of the window is **not** fixed, and it can be increased or decreased based on certain conditions.

For example, given a string, find the longest substring with unique characters.

Dynamic Sliding Window

```
s = "abcbad"
```

```
longest = 0
```

Dynamic Sliding Window

```
s = “abcbad”
```

```
longest = 1
```

Africa To Silicon Valley

Dynamic Sliding Window

```
s = "abcbad"
```

`longest = 2`

Dynamic Sliding Window

```
s = "abcbad"
```

```
longest = 3
```

Dynamic Sliding Window

```
s = "abcbad"
```

```
longest = 3  // duplicate found, shrink window
```

Dynamic Sliding Window

```
s = "abcbad"
```

```
longest = 3  // duplicate found, shrink window
```

Africa To Silicon Valley

Dynamic Sliding Window

```
s = “abcbad”
```

```
longest = 3
```

A2SV
Africa To Silicon Valley

Dynamic Sliding Window

s = “abcbad”

longest = 3

Dynamic Sliding Window

s = “ab<u>cbad</u>”

```
longest = 4
```

Longest substring without repeating character

Problem Link

Time and space complexity?

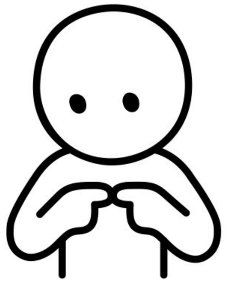
Dynamic sliding window

- Time complexity: **O(n)**
- Space complexity: **O(1)**

Dynamic sliding window

Let's look at another question on dynamic sliding window.

[Problem Link]



Common Pitfalls

Pitfalls

There are several common pitfalls when using sliding windows:

1. Off-by-one error
2. Not considering all possible cases
3. Not handling edge cases
4. Inefficient window updates
5. Not understanding the problem requirements

Off-by-one errors

An off-by-one error can occur when the window size is m, but the loop that iterates over the elements in the array uses an index range of [i, i+m), instead of [i, i+m-1).

For example, let's say you have an array of integers and you want to find the maximum sum of any subarray of length 3 (i.e., a sliding window of size 3).

Not considering all possible cases

There are often different cases to consider in each iteration of the algorithm. For example, if we are finding a maximum subarray sum of size k, we need to consider what happens

- ➢ If there are negative numbers in the array.

For an array that contains only positive numbers, the maximum subarray sum will always be the sum of the k largest elements in the array. However, if there are negative numbers in the array, that may not be the case.

Not handling edge cases

Failing to handle edge cases can result in incorrect output or runtime errors. Some common examples of edge cases that may need to be handled in a sliding window algorithm include:

➢ The window size is larger than the size of the data structure being processed.

➢ The data structure being processed is empty.

➢ The window size is 1 or 0.

➢ The input data contains negative numbers or is otherwise not in the expected format.

Inefficient Window Updates

This can happen when we are trying to update the window by moving it forward by one or more positions. If we do this by simply copying the elements of the window to the new position, it can result in a lot of redundant operations.

For example, suppose we have a window of size k and we want to move it one position to the right. If we simply copy all the elements of the window to the new position, we will end up copying (k-1) elements that are already in the new position. This can become very inefficient when k is large and/or the number of window updates is high.

Not understanding the problem requirements

This can lead to suboptimal or incorrect solutions. It's important to carefully read and understand the problem requirements and constraints. It's also important to think about edge cases and design the sliding window accordingly.

Complexity Analysis

- Fixed-size sliding window
  - Time complexity: **O(n)**
  - Space complexity: **O(1)**
- Dynamic sliding window
  - Time complexity: **O(n)**
  - Space complexity: **O(1)**
- Counting sliding window
  - Time complexity: **O(n)**
  - Space complexity: **O(1)**
- Two-dimensional sliding window
  - Time complexity: **O(n*m)**
  - Space complexity: **O(1)**

Applications of sliding window technique

- Finding minimum/maximum value from a list/array
- Finding the longest/shortest value from a list/array
- Applied on an ordered data structure
- Finding a target element from a list/array
- Finding selected sized pair of elements
- Calculating average

Practice Questions

- ➢ Minimum Size Subarray Sum
- ➢ Longest Repeating Character Replacement
- ➢ Permutation in String
- ➢ Smallest Range Covering Elements from K Lists
- ➢ Shortest Subarray with Sum at Least K
- ➢ Fruit Into Baskets
- ➢ Longest Turbulent Subarray
- ➢ Get Equal Substrings Within Budget
- ➢ Minimum Window Substring

A2SV Africa To Silicon Valley

Quote Of The Day

"Your big opportunity may be right where you are now. Open your eyes, and you'll find it in your windows."

- Napoleon Hill