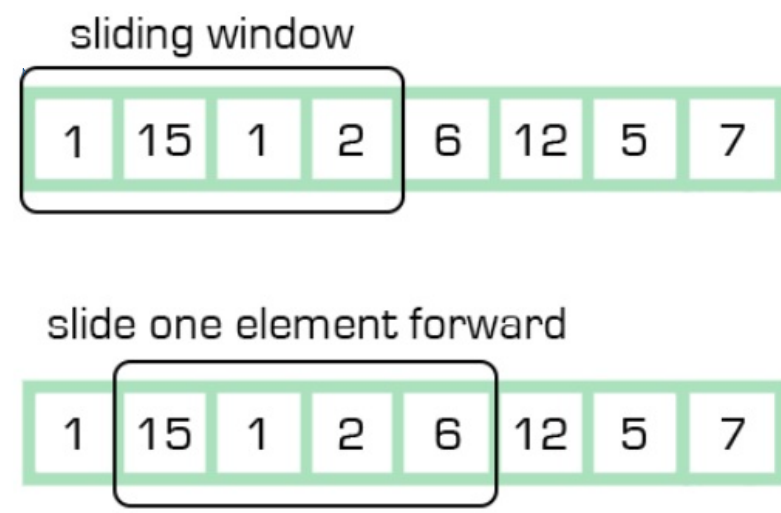


Sliding Window Technique

① Two pointer → [pair | Topdeal | Reverse]

① same direction

② opposite direction

Some Case → Array must be sorted

⑦ Sliding Window → Arrays or String

Arrays

Strings

SubArray → memory Allocation → continous

(I) Indexing

(II) continous memory Allocation

Substring → memory Allocation continous

point

sub

Substring / Subarray → They are just small part of array and string in continous manner.

Arr = [1, 2, 3, 4, 5] → [1, ~~3~~, 5]

String str = "World" → "Worl"

~~"Wd"~~

⇒

Sliding Window → Arrays or Strings

Subarray

Substring

Arr = [1, 2, 3, 4, 5] → All possible subarrays

$n = 5$

subarrays

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

[2], [2,3], [2,3,4], [2,3,4,5]

[3], [3,4], [3,4,5]

[4], [4,5]

[5]

$\rightarrow \frac{n(n+1)}{2}$

$\frac{5 \times (5+1)}{2} = 15$

(n^2)

String = "abc"

$\rightarrow \frac{n(n+1)}{2}$

$n = 3$

"a", "ab", "abc"

"b", "bc"

"c"

$\frac{n(n+1)}{2}$

$\frac{3 \times (3+1)}{2} \Rightarrow 6$

Arr = [1, 2, 3]

Brute

print All possible subarrays

[[1], [1,2], [1,2,3]
[2], [2,3]
[3]]

Arr ←

1	2	3
0	1	2

$n = 3$

Banda

$i = j$

Index

Index

$i <= j$

Brute force

$i = 0$

$j = 0$

[1], [1,2], [1,2,3]

[2], [2,3]

[3]

Before **Sliding Window:**

map and **unordered_map**

<map>

map

key-value → BST → (log n) → insert, delete

→ Sorted by key

key-value → HashTable → O(1)

Hashing

→ Not in any specific order

<unordered_map>

unordered_map

Feature	`map`	`unordered_map`
Ordering	Keys are stored in **sorted order** (ascending by default using `<`).	Keys have **no specific order**.
Underlying Data Structure	Self-balancing BST (usually Red-Black Tree)	Hash Table
Time Complexity	`O(log n)` for insert, delete, find	Average case `O(1)`, Worst case `O(n)`
Header File	`<map>`	`<unordered_map>`
Key Requirements	Must be **comparable** (`<` operator)	Must be **hashable** (`std::hash`)
Traversal	In sorted order by key	In arbitrary (unordered) order

Sliding Window Technique?

The **Sliding Window** is a **subarray** or **subset-based** technique that is mainly used for problems involving **arrays or strings**.

Instead of checking every possible subarray (which leads to $O(n^2)$ time), **Sliding Window** allows you to **move a window of size** `k` **across the array**, doing the needed calculation in $O(n)$ time.

Worst → Best

Brute → Optimal

When to Use Sliding Window?

Clue in Problem Statement	Example Phrases
Fixed/Variable-sized Subarray or Substring	"Find max sum of `k` consecutive elements", "Longest substring with at most `k` distinct characters"
Need to check or optimize values over subarrays	"Find minimum length of substring", "Count substrings with conditions"
Continuous elements	"Consecutive", "Contiguous", "Subarray"

Sliding Window

Easy

object / Hashmap / map / Dictionary

Model 1

Size of S.W. is **fixed**
Note: Given in question.

1, 2, 3, 4, 5

k = 3

Model 2

Size of S.W. is **variable**
Note: Not Given in question.

Array	1	2	3	4	5	6	7	8	9	10
Index	0	1	2	3	4	5	6	7	8	9
n-1										

Size =>

j - i + 1

7 - 3 + 1 = 5

100 → 98% → Studying Lo Indian

Fixed Sized **Sliding Window**

1. **Maximum Sum Subarray of Size K**

Given an array of integers and a number k, find the maximum sum of any contiguous subarray of size k.

Input: arr = [2, 1, 5, 1, 3, 2], k = 3
Output: 9

Brute force

↓

Sliding

[2, 1, 5, 1, 3, 2], k = 3

Subarray = k

sum = 8

sum = 7

sum ≠ 9

sum = 6

Using Brute Force

```
int maxSumK(int arr[], int n, int k) {
    int maxSum = 0;

    for (int i = 0; i <= n - k; i++) {
        int currentSum = 0;
        for (int j = i; j < i + k; j++) {
            currentSum += arr[j];
        }
        maxSum = max(maxSum, currentSum);
    }

    return maxSum;
}
```

$n - 6 - 3 \Rightarrow 3$

[2, 1, 5, 1, 3, 2], k = 3

0 1 2 3 4 5

$n * k \Rightarrow O(n * k)$

n

n^2

maxSum = 8

WS = 8 + 1 → 9 − 2 → 7

Using Sliding Window

```
int maxSumK(int arr[], int n, int k) {
    int maxSum = 0, windowSum = 0;

    for (int i = 0; i < k; i++)
        windowSum += arr[i];

    maxSum = windowSum;
    maxSum = max(maxSum, windowSum);
    for (int i = k; i < n; i++) {
        windowSum += arr[i] - arr[i - k];
        maxSum = max(maxSum, windowSum);
    }

    return maxSum;
}
```

[2, 1, 5, 1, 3, 2], k = 3

0 1 2 3 4 5

O(n)

2. **Count the Number of Vowels in All Substrings of Size k**

Given a string s and an integer k, count the number of vowels in every contiguous substring of size k.

Input: s = "abciiidef", k = 3
Output:

1 1 2

Using Brute Force

```
bool isVowel(char c) {
    c = tolower(c);
    return c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u';
}

void countVowelsBrute(string s, int k) {
    int n = s.length();

    for (int i = 0; i <= n - k; i++) {
        int count = 0;
        for (int j = i; j < i + k; j++) {
            if (isVowel(s[j])) count++;
        }
        cout << count << " ";
    }
}
```

Using Sliding Window

```
void countVowelsSliding(string s, int k) {
    int n = s.length();
    int count = 0;

    for (int i = 0; i < k; i++) {
        if (isVowel(s[i])) count++;
    }

    cout << count << " ";

    for (int i = k; i < n; i++) {
        if (isVowel(s[i - k])) count--;  // remove char going out
        if (isVowel(s[i])) count++;      // add char coming in
        cout << count << " ";
    }
}
```

Variable Size **Sliding Window**

Find is there any sub-array with the given sum [return True/ False]

	0	1	2	3	4	5
arr	1	4	20	3	10	5

$n = 6$

$sum = 33$

```
function fun(arr,n,sum) // variable size S.W
{
  windowSum=0, high=0
  for(low=0;low<n;low++)
  {
    while(windowSum<sum && high<n)
    {
      windowSum=windowSum+arr[high]
      high++
    }
    if(windowSum==sum) // happy
    {
      return true
    }
    windowSum=windowSum-arr[low]
  }
  return false
}
```

Find the size of largest sub-string which doesn't contains any repeated characters in given string

	0	1	2	3	4	5	6	7	8
s	a	b	b	c	d	e	a	b	b

$n = 9$

```
function longestUniqueSubsttr(s,n)
 {

   let hm be a hashmap/ object

   maximum_length = 0;


   start = 0;

   for(i= 0; i < n; i++)
   {

     if(hm.containsKey(s[i]))
     {
       start = Math.max(start, hm.get(s[i]) + 1);
     }
    hm.put(s[i], i);
     maximum_length = Math.max(maximum_length, i-start + 1);

   }
   return maximum_length;
 }
```

Find the Longest Substring which contains K distinct / Unique characters

	0	1	2	3	4	5	6	7	8	9	10
s	a	a	b	a	c	b	e	b	e	b	e

n = 11

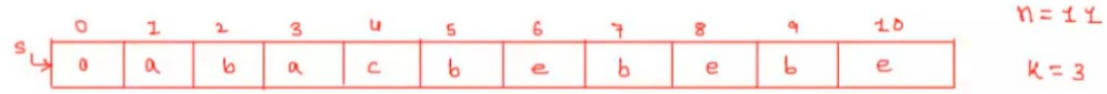
k = 3

```
int longestKDistinctSubstring(string s, int k) {
    unordered_map<char, int> freq;
    int start = 0, maxLen = 0;
    for (int end = 0; end < s.length(); end++) {
        freq[s[end]]++;

        while (freq.size() > k) {
            freq[s[start]]--;
            if (freq[s[start]] == 0)
                freq.erase(s[start]);
            start++;
        }

        if (freq.size() == k) {
            maxLen = max(maxLen, end - start + 1);
        }
    }

    return maxLen;
}
```



End