**Introduction**

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

Given an array, find the average of all subarrays of ‘K’ contiguous elements 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 subarrays of ‘5’ contiguous elements in the given array. Let’s solve this:

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

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

Output: [2.2, 2.8, 2.4, 3.6, 2.8]

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

import java.util.Arrays;

class AverageOfSubarrayOfSizeK {

public static double[] findAverages(int K, int[] arr) {

double[] result = new double[arr.length - K + 1];

for (int i = 0; i <= arr.length - K; i++) {

// find sum of next 'K' elements

double sum = 0;

for (int j = i; j < i + K; j++)

sum += arr[j];

result[i] = sum / K; // calculate average

}

return result;

}

public static void main(String[] args) {

double[] result = AverageOfSubarrayOfSizeK.findAverages(5, new int[] { 1, 3, 2, 6, -1, 4, 1, 8, 2 });

System.out.println("Averages of subarrays of size K: " + Arrays.toString(result));

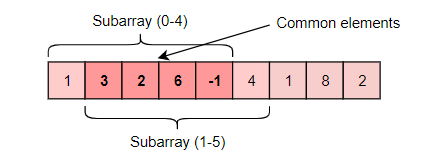
}

}

**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)*O*(*N*∗*K*) where ‘N’ is the number of elements in the input array.

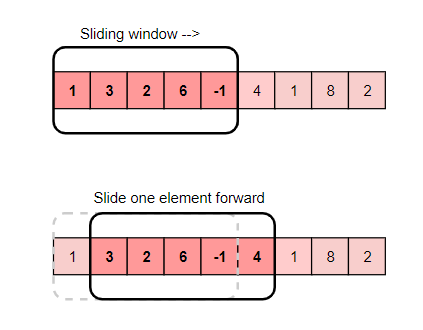
Can we find a better solution? Do you see any inefficiency in the above approach?

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 subarray as a sliding window of ‘5’ elements. This means that we will slide the window by one element when we move on to the next subarray. 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)*O*(*N*).



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

import java.util.Arrays;

class AverageOfSubarrayOfSizeK {

public static double[] findAverages(int K, int[] arr) {

double[] result = new double[arr.length - K + 1];

double windowSum = 0;

int windowStart = 0;

for (int windowEnd = 0; windowEnd < arr.length; windowEnd++) {

windowSum += arr[windowEnd]; // add the next element

// slide the window, we don't need to slide if we've not hit the required window size of 'k'

if (windowEnd >= K - 1) {

result[windowStart] = windowSum / K; // calculate the average

windowSum -= arr[windowStart]; // subtract the element going out

windowStart++; // slide the window ahead

}

}

return result;

}

public static void main(String[] args) {

double[] result = AverageOfSubarrayOfSizeK.findAverages(5, new int[] { 1, 3, 2, 6, -1, 4, 1, 8, 2 });

System.out.println("Averages of subarrays of size K: " + Arrays.toString(result));

}

}

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.

**Maximum Sum Subarray of Size K (easy)**

**We'll cover the following**

* + [Problem Statement](https://www.educative.io/courses/grokking-the-coding-interview/JPKr0kqLGNP#Problem-Statement)
  + [Try it yourself](https://www.educative.io/courses/grokking-the-coding-interview/JPKr0kqLGNP#Try-it-yourself)
  + [Solution](https://www.educative.io/courses/grokking-the-coding-interview/JPKr0kqLGNP#Solution)
    - [Code](https://www.educative.io/courses/grokking-the-coding-interview/JPKr0kqLGNP#Code)
    - [A better approach](https://www.educative.io/courses/grokking-the-coding-interview/JPKr0kqLGNP#A-better-approach)
    - [Time Complexity](https://www.educative.io/courses/grokking-the-coding-interview/JPKr0kqLGNP#Time-Complexity)
    - [Space Complexity](https://www.educative.io/courses/grokking-the-coding-interview/JPKr0kqLGNP#Space-Complexity)

**Problem Statement**[#](https://www.educative.io/courses/grokking-the-coding-interview/JPKr0kqLGNP#Problem-Statement)

Given an array of positive numbers and a positive number ‘k,’ find the **maximum sum of any contiguous subarray of size ‘k’**.

**Example 1:**

Input: [2, 1, 5, 1, 3, 2], k=3   
Output: 9  
Explanation: Subarray with maximum sum is [5, 1, 3].

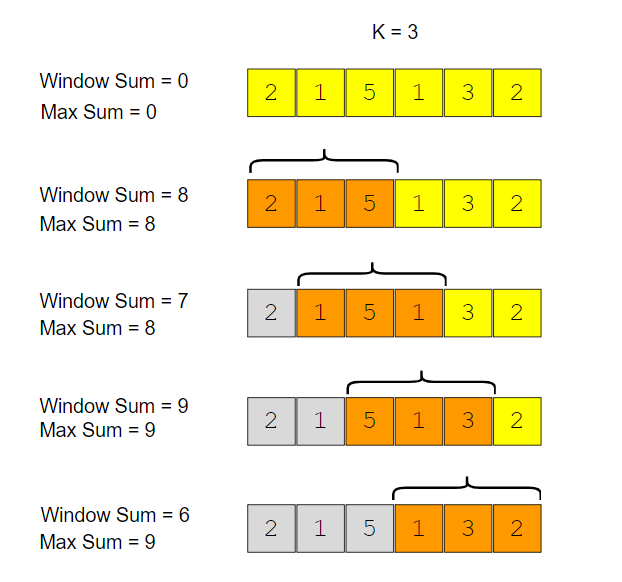
**Example 2:**

Input: [2, 3, 4, 1, 5], k=2   
Output: 7  
Explanation: Subarray with maximum sum is [3, 4].

**Try it yourself**[#](https://www.educative.io/courses/grokking-the-coding-interview/JPKr0kqLGNP#Try-it-yourself)

## Solution[#](https://www.educative.io/courses/grokking-the-coding-interview/JPKr0kqLGNP#Solution)

A basic brute force solution will be to calculate the sum of all ‘k’ sized subarrays of the given array to find the subarray with the highest sum. We can start from every index of the given array and add the next ‘k’ elements to find the subarray’s sum. Following is the visual representation of this algorithm for Example-1:



### Code[#](https://www.educative.io/courses/grokking-the-coding-interview/JPKr0kqLGNP#Code)

Here is what our algorithm will look like:

class MaxSumSubArrayOfSizeK {

public static int findMaxSumSubArray(int k, int[] arr) {

int maxSum = 0, windowSum;

for (int i = 0; i <= arr.length - k; i++) {

windowSum = 0;

for (int j = i; j < i + k; j++) {

windowSum += arr[j];

}

maxSum = Math.max(maxSum, windowSum);

}

return maxSum;

}

public static void main(String[] args) {

System.out.println("Maximum sum of a subarray of size K: "

+ MaxSumSubArrayOfSizeK.findMaxSumSubArray(3, new int[] { 2, 1, 5, 1, 3, 2 }));

System.out.println("Maximum sum of a subarray of size K: "

+ MaxSumSubArrayOfSizeK.findMaxSumSubArray(2, new int[] { 2, 3, 4, 1, 5 }));

}

}

The above algorithm’s time complexity will be O(N\*K)*O*(*N*∗*K*), where ‘N’ is the total number of elements in the given array. Is it possible to find a better algorithm than this?

**A better approach**[#](https://www.educative.io/courses/grokking-the-coding-interview/JPKr0kqLGNP#A-better-approach)

If you observe closely, you will realize that to calculate the sum of a contiguous subarray, we can utilize the sum of the previous subarray. For this, consider each subarray as a **Sliding Window** of size ‘k.’ To calculate the sum of the next subarray, we need to slide the window ahead by one element. So to slide the window forward and calculate the sum of the new position of the sliding window, we need to do two things:

1. Subtract the element going out of the sliding window, i.e., subtract the first element of the window.
2. Add the new element getting included in the sliding window, i.e., the element coming right after the end of the window.

This approach will save us from re-calculating the sum of the overlapping part of the sliding window. Here is what our algorithm will look like:

class MaxSumSubArrayOfSizeK {

public static int findMaxSumSubArray(int k, int[] arr) {

int windowSum = 0, maxSum = 0;

int windowStart = 0;

for (int windowEnd = 0; windowEnd < arr.length; windowEnd++) {

windowSum += arr[windowEnd]; // add the next element

// slide the window, we don't need to slide if we've not hit the required window size of 'k'

if (windowEnd >= k - 1) {

maxSum = Math.max(maxSum, windowSum);

windowSum -= arr[windowStart]; // subtract the element going out

windowStart++; // slide the window ahead

}

}

return maxSum;

}

public static void main(String[] args) {

System.out.println("Maximum sum of a subarray of size K: "

+ MaxSumSubArrayOfSizeK.findMaxSumSubArray(3, new int[] { 2, 1, 5, 1, 3, 2 }));

System.out.println("Maximum sum of a subarray of size K: "

+ MaxSumSubArrayOfSizeK.findMaxSumSubArray(2, new int[] { 2, 3, 4, 1, 5 }));

}

}

### Time Complexity[#](https://www.educative.io/courses/grokking-the-coding-interview/JPKr0kqLGNP#Time-Complexity)

The time complexity of the above algorithm will be O(N)*O*(*N*).

### Space Complexity[#](https://www.educative.io/courses/grokking-the-coding-interview/JPKr0kqLGNP#Space-Complexity)

The algorithm runs in constant space O(1)*O*(1).

**Smallest Subarray With a Greater Sum (easy)**

**We'll cover the following**

* + [Problem Statement](https://www.educative.io/courses/grokking-the-coding-interview/7XMlMEQPnnQ#Problem-Statement)
  + [Try it yourself](https://www.educative.io/courses/grokking-the-coding-interview/7XMlMEQPnnQ#Try-it-yourself)
  + [Solution](https://www.educative.io/courses/grokking-the-coding-interview/7XMlMEQPnnQ#Solution)
  + [Code](https://www.educative.io/courses/grokking-the-coding-interview/7XMlMEQPnnQ#Code)
    - [Time Complexity](https://www.educative.io/courses/grokking-the-coding-interview/7XMlMEQPnnQ#Time-Complexity)
    - [Space Complexity](https://www.educative.io/courses/grokking-the-coding-interview/7XMlMEQPnnQ#Space-Complexity)

**Problem Statement**[#](https://www.educative.io/courses/grokking-the-coding-interview/7XMlMEQPnnQ#Problem-Statement)

Given an array of positive numbers and a positive number ‘S,’ find the length of the **smallest** contiguous subarray whose sum is **greater than or equal to ‘S’**. Return 0 if no such subarray exists.

**Example 1:**

Input: [2, 1, 5, 2, 3, 2], S=7   
Output: 2  
Explanation: The smallest subarray with a sum greater than or equal to '7' is [5, 2].

**Example 2:**

Input: [2, 1, 5, 2, 8], S=7   
Output: 1  
Explanation: The smallest subarray with a sum greater than or equal to '7' is [8].

**Example 3:**

Input: [3, 4, 1, 1, 6], S=8   
Output: 3  
Explanation: Smallest subarrays with a sum greater than or equal to '8' are [3, 4, 1]   
or [1, 1, 6].

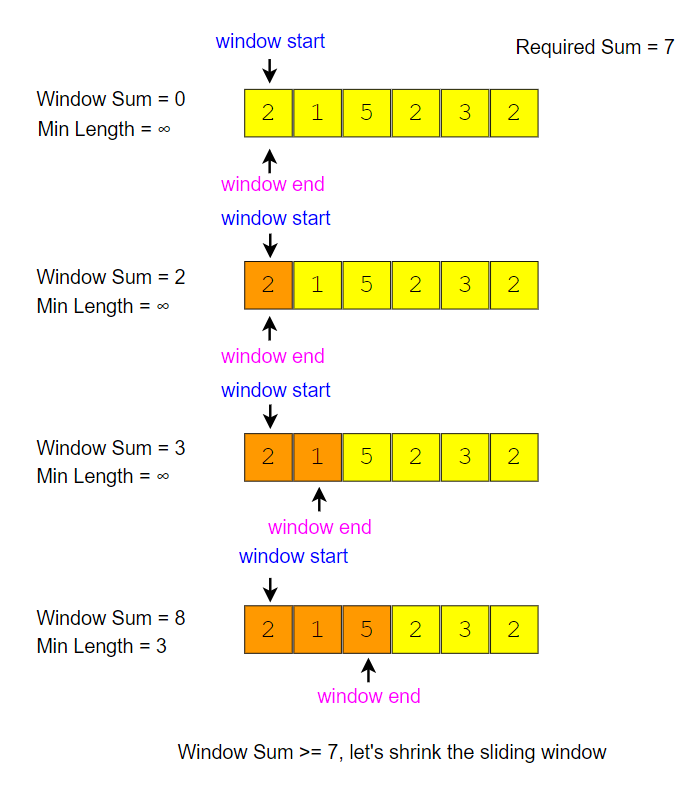
**Try it yourself**[#](https://www.educative.io/courses/grokking-the-coding-interview/7XMlMEQPnnQ#Try-it-yourself)

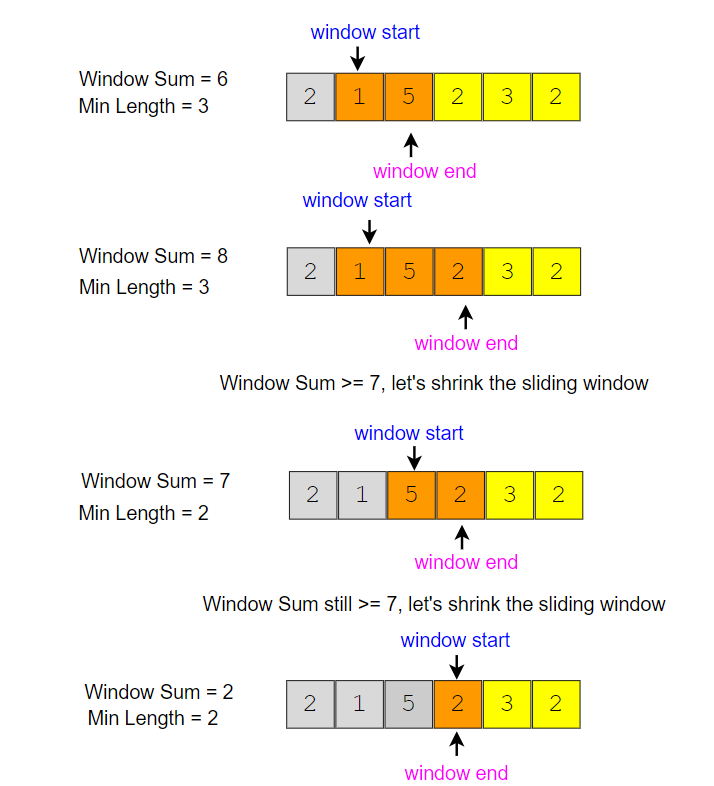
**Solution**[#](https://www.educative.io/courses/grokking-the-coding-interview/7XMlMEQPnnQ#Solution)

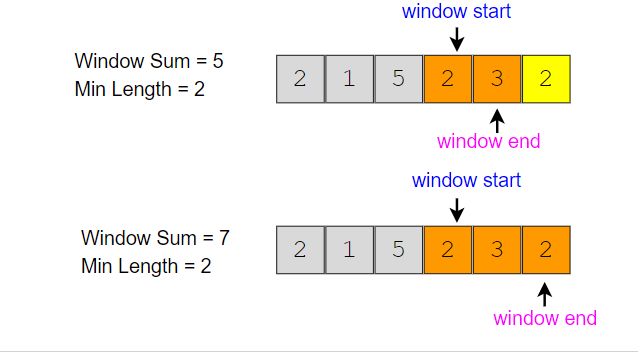
This problem follows the **Sliding Window** pattern, and we can use a similar strategy as discussed in [Maximum Sum Subarray of Size K](https://www.educative.io/collection/page/5668639101419520/5671464854355968/5177043027230720/). There is one difference though: in this problem, the sliding window size is not fixed. Here is how we will solve this problem:

1. First, we will add-up elements from the beginning of the array until their sum becomes greater than or equal to ‘S.’
2. These elements will constitute our sliding window. We are asked to find the smallest such window having a sum greater than or equal to ‘S.’ We will remember the length of this window as the smallest window so far.
3. After this, we will keep adding one element in the sliding window (i.e., slide the window ahead) in a stepwise fashion.
4. In each step, we will also try to shrink the window from the beginning. We will shrink the window until the window’s sum is smaller than ‘S’ again. This is needed as we intend to find the smallest window. This shrinking will also happen in multiple steps; in each step, we will do two things:
   * Check if the current window length is the smallest so far, and if so, remember its length.
   * Subtract the first element of the window from the running sum to shrink the sliding window.

Here is the visual representation of this algorithm for the Example-1:







## Code[#](https://www.educative.io/courses/grokking-the-coding-interview/7XMlMEQPnnQ#Code)

Here is what our algorithm will look like:

class MinSizeSubArraySum {

public static int findMinSubArray(int S, int[] arr) {

int windowSum = 0, minLength = Integer.MAX\_VALUE;

int windowStart = 0;

for (int windowEnd = 0; windowEnd < arr.length; windowEnd++) {

windowSum += arr[windowEnd]; // add the next element

// shrink the window as small as possible until the 'windowSum' is smaller than 'S'

while (windowSum >= S) {

minLength = Math.min(minLength, windowEnd - windowStart + 1);

windowSum -= arr[windowStart]; // subtract the element going out

windowStart++; // slide the window ahead

}

}

return minLength == Integer.MAX\_VALUE ? 0 : minLength;

}

public static void main(String[] args) {

int result = MinSizeSubArraySum.findMinSubArray(7, new int[] { 2, 1, 5, 2, 3, 2 });

System.out.println("Smallest subarray length: " + result);

result = MinSizeSubArraySum.findMinSubArray(7, new int[] { 2, 1, 5, 2, 8 });

System.out.println("Smallest subarray length: " + result);

result = MinSizeSubArraySum.findMinSubArray(8, new int[] { 3, 4, 1, 1, 6 });

System.out.println("Smallest subarray length: " + result);

}

}

### Time Complexity[#](https://www.educative.io/courses/grokking-the-coding-interview/7XMlMEQPnnQ#Time-Complexity)

The time complexity of the above algorithm will be O(N)*O*(*N*). The outer for loop runs for all elements, and the inner while loop processes each element only once; therefore, the time complexity of the algorithm will be O(N+N)*O*(*N*+*N*), which is asymptotically equivalent to O(N)*O*(*N*).

### Space Complexity[#](https://www.educative.io/courses/grokking-the-coding-interview/7XMlMEQPnnQ#Space-Complexity)

The algorithm runs in constant space O(1)*O*(1).

**Longest Substring with maximum K Distinct Characters (medium)**

**We'll cover the following**

* + [Problem Statement](https://www.educative.io/courses/grokking-the-coding-interview/YQQwQMWLx80#Problem-Statement)
  + [Try it yourself](https://www.educative.io/courses/grokking-the-coding-interview/YQQwQMWLx80#Try-it-yourself)
  + [Solution](https://www.educative.io/courses/grokking-the-coding-interview/YQQwQMWLx80#Solution)
  + [Code](https://www.educative.io/courses/grokking-the-coding-interview/YQQwQMWLx80#Code)
    - [Time Complexity](https://www.educative.io/courses/grokking-the-coding-interview/YQQwQMWLx80#Time-Complexity)
    - [Space Complexity](https://www.educative.io/courses/grokking-the-coding-interview/YQQwQMWLx80#Space-Complexity)

**Problem Statement**[#](https://www.educative.io/courses/grokking-the-coding-interview/YQQwQMWLx80#Problem-Statement)

Given a string, find the length of the **longest substring** in it **with no more than K distinct characters**.

**Example 1:**

Input: String="araaci", K=2  
Output: 4  
Explanation: The longest substring with no more than '2' distinct characters is "araa".

**Example 2:**

Input: String="araaci", K=1  
Output: 2  
Explanation: The longest substring with no more than '1' distinct characters is "aa".

**Example 3:**

Input: String="cbbebi", K=3  
Output: 5  
Explanation: The longest substrings with no more than '3' distinct characters are "cbbeb" & "bbebi".

**Example 4:**

Input: String="cbbebi", K=10  
Output: 6  
Explanation: The longest substring with no more than '10' distinct characters is "cbbebi".

**Try it yourself**[#](https://www.educative.io/courses/grokking-the-coding-interview/YQQwQMWLx80#Try-it-yourself)

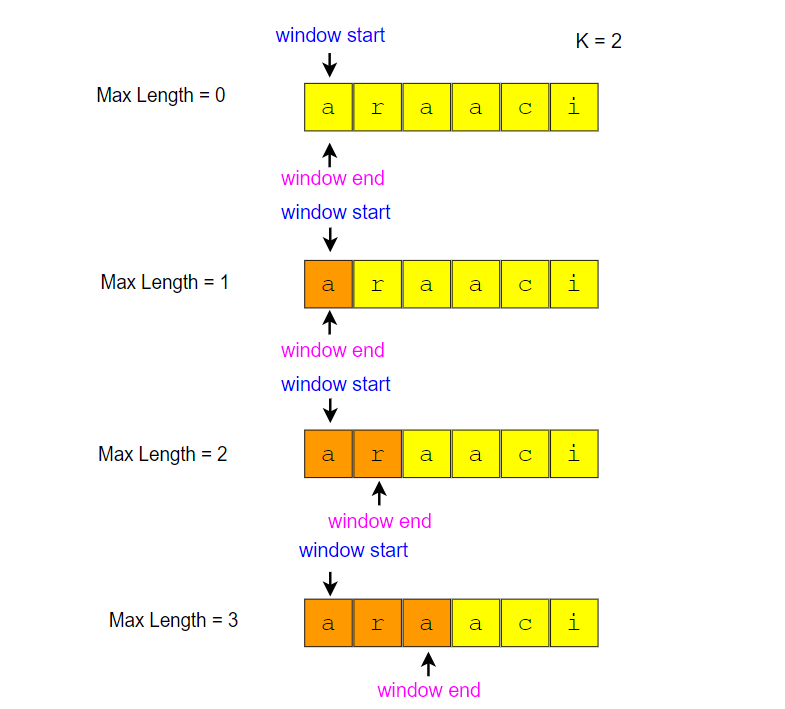
**Solution**[#](https://www.educative.io/courses/grokking-the-coding-interview/YQQwQMWLx80#Solution)

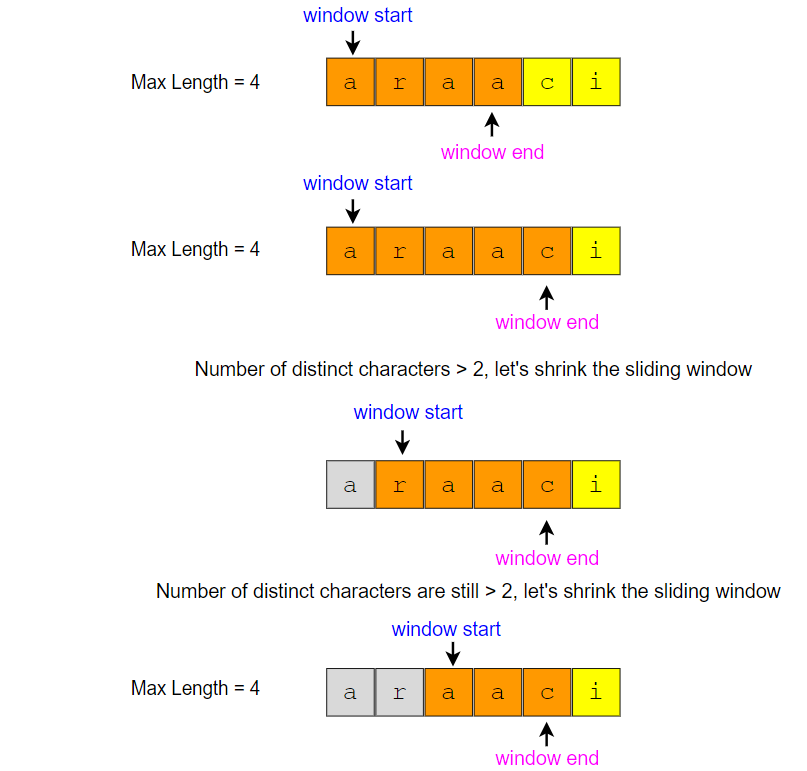
This problem follows the **Sliding Window** pattern, and we can use a similar dynamic sliding window strategy as discussed in [Smallest Subarray With a Greater Sum](https://www.educative.io/collection/page/5668639101419520/5671464854355968/5177043027230720/). We can use a **HashMap** to remember the frequency of each character we have processed. Here is how we will solve this problem:

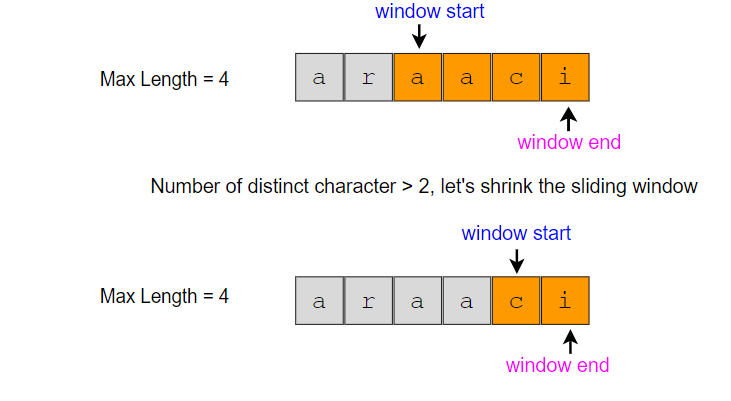
1. First, we will insert characters from the beginning of the string until we have K distinct characters in the **HashMap**.
2. These characters will constitute our sliding window. We are asked to find the longest such window having no more than K distinct characters. We will remember the length of this window as the longest window so far.
3. After this, we will keep adding one character in the sliding window (i.e., slide the window ahead) in a stepwise fashion.
4. In each step, we will try to shrink the window from the beginning if the count of distinct characters in the **HashMap** is larger than K. We will shrink the window until we have no more than K distinct characters in the **HashMap**. This is needed as we intend to find the longest window.
5. While shrinking, we’ll decrement the character’s frequency going out of the window and remove it from the **HashMap** if its frequency becomes zero.
6. At the end of each step, we’ll check if the current window length is the longest so far, and if so, remember its length.

Here is the visual representation of this algorithm for the Example-1:

window end window start window start window end window start window end window start window end window start window end window start window end window start window end window start window end window start window end a r a a c i K = 2 Max Length = 0 a r a a c i a r a a c i Max Length = 1 Max Length = 2 a r a a c i Max Length = 3 a r a a c i Max Length = 4 a r a a c i a r a a c i Max Length = 4 Max Length = 4 a r a a c i a r a a c i Max Length = 4 Max Length = 4 window start window end a r a a c i Number of distinct characters > 2, let's shrink the sliding window Number of distinct characters are still > 2, let's shrink the sliding window Number of distinct character > 2, let's shrink the sliding window







**Code**[#](https://www.educative.io/courses/grokking-the-coding-interview/YQQwQMWLx80#Code)

import java.util.\*;

class LongestSubstringKDistinct {

public static int findLength(String str, int k) {

if (str == null || str.length() == 0)

throw new IllegalArgumentException();

int windowStart = 0, maxLength = 0;

Map<Character, Integer> charFrequencyMap = new HashMap<>();

// in the following loop we'll try to extend the range [windowStart, windowEnd]

for (int windowEnd = 0; windowEnd < str.length(); windowEnd++) {

char rightChar = str.charAt(windowEnd);

charFrequencyMap.put(rightChar, charFrequencyMap.getOrDefault(rightChar, 0) + 1);

// shrink the sliding window, until we are left with 'k' distinct characters in the frequency map

while (charFrequencyMap.size() > k) {

char leftChar = str.charAt(windowStart);

charFrequencyMap.put(leftChar, charFrequencyMap.get(leftChar) - 1);

if (charFrequencyMap.get(leftChar) == 0) {

charFrequencyMap.remove(leftChar);

}

windowStart++; // shrink the window

}

maxLength = Math.max(maxLength, windowEnd - windowStart + 1); // remember the maximum length so far

}

return maxLength;

}

public static void main(String[] args) {

System.out.println("Length of the longest substring: " + LongestSubstringKDistinct.findLength("araaci", 2));

System.out.println("Length of the longest substring: " + LongestSubstringKDistinct.findLength("araaci", 1));

System.out.println("Length of the longest substring: " + LongestSubstringKDistinct.findLength("cbbebi", 3));

}

}

The above algorithm’s time complexity will be O(N), where N is the number of characters in the input string. The outer for loop runs for all characters, and the inner while loop processes each character only once; therefore, the time complexity of the algorithm will be O(N+N), which is asymptotically equivalent to O(N).

**Space Complexity**[#](https://www.educative.io/courses/grokking-the-coding-interview/YQQwQMWLx80#Space-Complexity)

The algorithm’s space complexity is O(K), as we will be storing a maximum of K+1 characters in the HashMap.

# Fruits into Baskets (medium)

**We'll cover the following**

* + [Problem Statement](https://www.educative.io/courses/grokking-the-coding-interview/Bn2KLlOR0lQ#Problem-Statement)
  + [Try it yourself](https://www.educative.io/courses/grokking-the-coding-interview/Bn2KLlOR0lQ#Try-it-yourself)
  + [Solution](https://www.educative.io/courses/grokking-the-coding-interview/Bn2KLlOR0lQ#Solution)
  + [Code](https://www.educative.io/courses/grokking-the-coding-interview/Bn2KLlOR0lQ#Code)
    - [Time Complexity](https://www.educative.io/courses/grokking-the-coding-interview/Bn2KLlOR0lQ#Time-Complexity)
    - [Space Complexity](https://www.educative.io/courses/grokking-the-coding-interview/Bn2KLlOR0lQ#Space-Complexity)
  + [Similar Problems](https://www.educative.io/courses/grokking-the-coding-interview/Bn2KLlOR0lQ#Similar-Problems)

## Problem Statement[#](https://www.educative.io/courses/grokking-the-coding-interview/Bn2KLlOR0lQ#Problem-Statement)

You are visiting a farm to collect fruits. The farm has a single row of fruit trees. You will be given two baskets, and your goal is to pick as many fruits as possible to be placed in the given baskets.

You will be given an array of characters where each character represents a fruit tree. The farm has following restrictions:

1. Each basket can have only one type of fruit. There is no limit to how many fruit a basket can hold.
2. You can start with any tree, but you can’t skip a tree once you have started.
3. You will pick exactly one fruit from every tree until you cannot, i.e., you will stop when you have to pick from a third fruit type.

Write a function to return the maximum number of fruits in both baskets.

**Example 1:**

Input: Fruit=['A', 'B', 'C', 'A', 'C']  
Output: 3  
Explanation: We can put 2 'C' in one basket and one 'A' in the other from the subarray ['C', 'A', 'C']

**Example 2:**

Input: Fruit=['A', 'B', 'C', 'B', 'B', 'C']  
Output: 5  
Explanation: We can put 3 'B' in one basket and two 'C' in the other basket.   
This can be done if we start with the second letter: ['B', 'C', 'B', 'B', 'C']

## Solution[#](https://www.educative.io/courses/grokking-the-coding-interview/Bn2KLlOR0lQ#Solution)

This problem follows the **Sliding Window** pattern and is quite similar to [Longest Substring with K Distinct Characters](https://www.educative.io/collection/page/5668639101419520/5671464854355968/5698217712812032/). In this problem, we need to find the length of the longest subarray with no more than two distinct characters (or fruit types!). This transforms the current problem into **Longest Substring with K Distinct Characters** where K=2.

## Code[#](https://www.educative.io/courses/grokking-the-coding-interview/Bn2KLlOR0lQ#Code)

Here is what our algorithm will look like, only the highlighted lines are different from [Longest Substring with K Distinct Characters](https://www.educative.io/collection/page/5668639101419520/5671464854355968/5698217712812032/):

import java.util.\*;

class MaxFruitCountOf2Types {

public static int findLength(char[] arr) {

int windowStart = 0, maxLength = 0;

Map<Character, Integer> fruitFrequencyMap = new HashMap<>();

// try to extend the range [windowStart, windowEnd]

for (int windowEnd = 0; windowEnd < arr.length; windowEnd++) {

fruitFrequencyMap.put(arr[windowEnd], fruitFrequencyMap.getOrDefault(arr[windowEnd], 0) + 1);

// shrink the sliding window, until we are left with '2' fruits in the frequency map

while (fruitFrequencyMap.size() > 2) {

fruitFrequencyMap.put(arr[windowStart], fruitFrequencyMap.get(arr[windowStart]) - 1);

if (fruitFrequencyMap.get(arr[windowStart]) == 0) {

fruitFrequencyMap.remove(arr[windowStart]);

}

windowStart++; // shrink the window

}

maxLength = Math.max(maxLength, windowEnd - windowStart + 1);

}

return maxLength;

}

public static void main(String[] args) {

System.out.println("Maximum number of fruits: " +

MaxFruitCountOf2Types.findLength(new char[] { 'A', 'B', 'C', 'A', 'C' }));

System.out.println("Maximum number of fruits: " +

MaxFruitCountOf2Types.findLength(new char[] { 'A', 'B', 'C', 'B', 'B', 'C' }));

}

}

### Time Complexity[#](https://www.educative.io/courses/grokking-the-coding-interview/Bn2KLlOR0lQ#Time-Complexity)

The above algorithm’s time complexity will be O(N), where ‘N’ is the number of characters in the input array. The outer for loop runs for all characters, and the inner while loop processes each character only once; therefore, the time complexity of the algorithm will be O(N+N), which is asymptotically equivalent to O(N).

### Space Complexity[#](https://www.educative.io/courses/grokking-the-coding-interview/Bn2KLlOR0lQ#Space-Complexity)

The algorithm runs in constant space O(1) as there can be a maximum of three types of fruits stored in the frequency map.

## Similar Problems[#](https://www.educative.io/courses/grokking-the-coding-interview/Bn2KLlOR0lQ#Similar-Problems)

**Problem 1: Longest Substring with at most 2 distinct characters**

Given a string, find the length of the longest substring in it with at most two distinct characters.

**Solution:** This problem is exactly similar to our parent problem.

# Longest Substring with Distinct Characters (hard)

**We'll cover the following**

* + [Problem Statement](https://www.educative.io/courses/grokking-the-coding-interview/YMzBx1gE5EO#Problem-Statement)
  + [Try it yourself](https://www.educative.io/courses/grokking-the-coding-interview/YMzBx1gE5EO#Try-it-yourself)
  + [Solution](https://www.educative.io/courses/grokking-the-coding-interview/YMzBx1gE5EO#Solution)
  + [Code](https://www.educative.io/courses/grokking-the-coding-interview/YMzBx1gE5EO#Code)
    - [Time Complexity](https://www.educative.io/courses/grokking-the-coding-interview/YMzBx1gE5EO#Time-Complexity)
    - [Space Complexity](https://www.educative.io/courses/grokking-the-coding-interview/YMzBx1gE5EO#Space-Complexity)

## Problem Statement[#](https://www.educative.io/courses/grokking-the-coding-interview/YMzBx1gE5EO#Problem-Statement)

Given a string, find the **length of the longest substring**, which has all **distinct characters**.

**Example 1:**

Input: String="aabccbb"  
Output: 3  
Explanation: The longest substring with distinct characters is "abc".

**Example 2:**

Input: String="abbbb"  
Output: 2  
Explanation: The longest substring with distinct characters is "ab".

**Example 3:**

Input: String="abccde"  
Output: 3  
Explanation: Longest substrings with distinct characters are "abc" & "cde".

## Solution[#](https://www.educative.io/courses/grokking-the-coding-interview/YMzBx1gE5EO#Solution)

This problem follows the **Sliding Window** pattern, and we can use a similar dynamic sliding window strategy as discussed in [Longest Substring with K Distinct Characters](https://www.educative.io/collection/page/5668639101419520/5671464854355968/5698217712812032/). We can use a **HashMap** to remember the last index of each character we have processed. Whenever we get a duplicate character, we will shrink our sliding window to ensure that we always have distinct characters in the sliding window.

## Code[#](https://www.educative.io/courses/grokking-the-coding-interview/YMzBx1gE5EO#Code)

Here is what our algorithm will look like:

import java.util.\*;

class NoRepeatSubstring {

public static int findLength(String str) {

int windowStart = 0, maxLength = 0;

Map<Character, Integer> charIndexMap = new HashMap<>();

// try to extend the range [windowStart, windowEnd]

for (int windowEnd = 0; windowEnd < str.length(); windowEnd++) {

char rightChar = str.charAt(windowEnd);

// if the map already contains the 'rightChar', shrink the window from the beginning so that

// we have only one occurrence of 'rightChar'

if (charIndexMap.containsKey(rightChar)) {

// this is tricky; in the current window, we will not have any 'rightChar' after its previous index

// and if 'windowStart' is already ahead of the last index of 'rightChar', we'll keep 'windowStart'

windowStart = Math.max(windowStart, charIndexMap.get(rightChar) + 1);

}

charIndexMap.put(rightChar, windowEnd); // insert the 'rightChar' into the map

maxLength = Math.max(maxLength, windowEnd - windowStart + 1); // remember the maximum length so far

}

return maxLength;

}

public static void main(String[] args) {

System.out.println("Length of the longest substring: " + NoRepeatSubstring.findLength("aabccbb"));

System.out.println("Length of the longest substring: " + NoRepeatSubstring.findLength("abbbb"));

System.out.println("Length of the longest substring: " + NoRepeatSubstring.findLength("abccde"));

}

}

### Time Complexity[#](https://www.educative.io/courses/grokking-the-coding-interview/YMzBx1gE5EO#Time-Complexity)

The above algorithm’s time complexity will be O(N), where ‘N’ is the number of characters in the input string.

### Space Complexity[#](https://www.educative.io/courses/grokking-the-coding-interview/YMzBx1gE5EO#Space-Complexity)

The algorithm’s space complexity will be O(K), where K is the number of distinct characters in the input string. This also means K<=N, because in the worst case, the whole string might not have any duplicate character, so the entire string will be added to the **HashMap**. Having said that, since we can expect a fixed set of characters in the input string (e.g., 26 for English letters), we can say that the algorithm runs in fixed space O(1); in this case, we can use a fixed-size array instead of the **HashMap**.