Sliding Window

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:

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

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:

Java | Python3 | C++ | JS

```cpp
using namespace std;

#include <iostream>
#include <vector>

class AverageOfSubarrayOfSizeK {
 public:
  static vector<double> findAverages(int K, const vector<int>& arr) {
    vector<double> result(arr.size() - K + 1);
    for (int i = 0; i <= arr.size() - 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;
  }
};

int main(int argc, char* argv[]) {
  vector<double> result =
      AverageOfSubarrayOfSizeK::findAverages(5, vector<int>{1, 3, 2, 6, -1, 4, 1, 8, 2});
  cout << "Averages of subarrays of size K: ";
  for (double d : result) {
    cout << d << " ";
  }
  cout << endl;
}

```

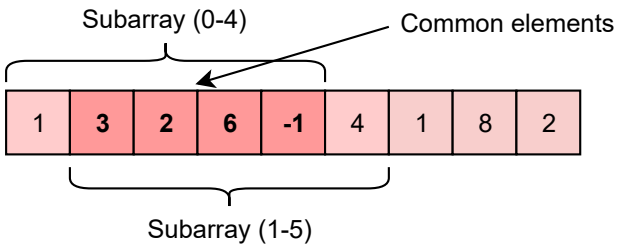
Output 0.941s

```
Averages of subarrays of size K: 2.2 2.8 2.4 3.6 2.8
```

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.

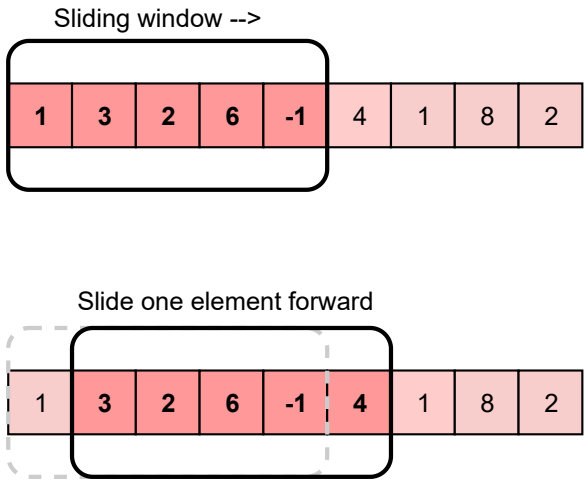
Can we find a better solution? Do you see any inefficiency in the above approach?

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 $O(N)$.



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

Java | Python3 | C++ | JS

```cpp
using namespace std;

#include <iostream>
#include <vector>

class AverageOfSubarrayOfSizeK {
 public:
  static vector<double> findAverages(int K, const vector<int>& arr) {
    vector<double> result(arr.size() - K + 1);
    double windowSum = 0;
    int windowStart = 0;
    for (int windowEnd = 0; windowEnd < arr.size(); 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;
  }
};

int main(int argc, char* argv[]) {
  vector<double> result =
      AverageOfSubarrayOfSizeK::findAverages(5, vector<int>{1, 3, 2, 6, -1, 4, 1, 8, 2});
  cout << "Averages of subarrays of size K: ";
  for (double d : result) {
    cout << d << " ";
  }
  cout << endl;
}

```

Output 0.904s

```
Averages of subarrays of size K: 2.2 2.8 2.4 3.6 2.8
```

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)

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

Try solving this question here:

Java | Python3 | JS | C++

```
using namespace std;

#include <iostream>
#include <vector>

class MaxSumSubArrayOfSizeK {
 public:
  static int findMaxSumSubArray(int k, const vector<int>& arr) {
    int maxSum = 0;
    // TODO: Write your code here
    return maxSum;
  }
};

```

Show Results | Show Console

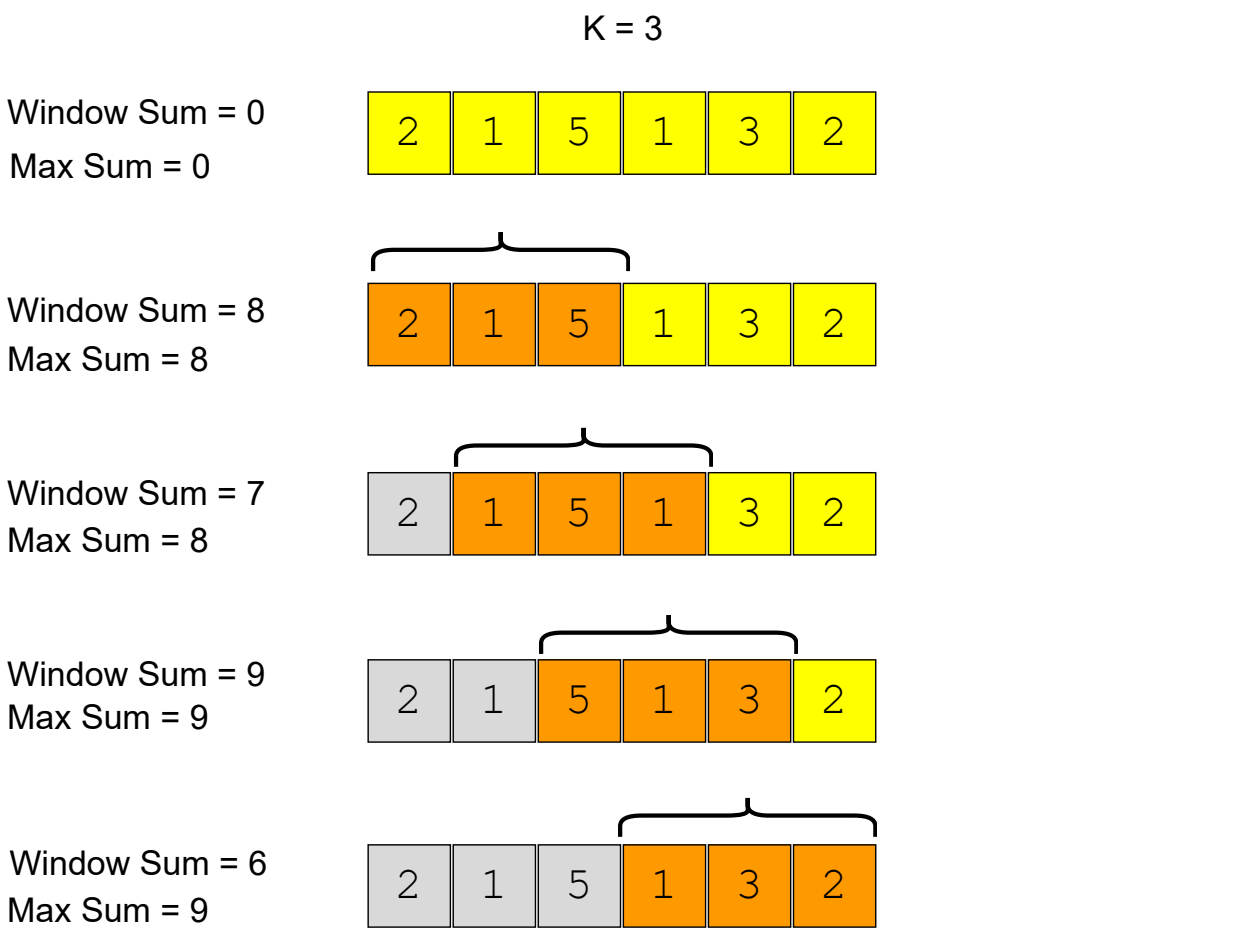
0 of 2 Tests Passed

Result	Input	Expected Output	Actual Output	Reason
✗	max_sub_array_of_size_k(3, [2, 1, 5, 1, ...	9	-1	Incorrect Output
✗	max_sub_array_of_size_k(2, [2, 3, 4, 1, ...	7	-1	Incorrect Output

0.496s

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 sum of the subarray. Following is the visual representation of this algorithm for Example-1:



Code

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```cpp
using namespace std;

#include <iostream>
#include <vector>

class MaxSumSubArrayOfSizeK {
 public:
  static int findMaxSumSubArray(int k, const vector<int>& arr) {
    int maxSum = 0, windowSum;
    for (int i = 0; i <= arr.size() - k; i++) {
      windowSum = 0;
      for (int j = i; j < i + k; j++) {
        windowSum += arr[j];
      }
      maxSum = max(maxSum, windowSum);
    }

    return maxSum;
  }
};

int main(int argc, char* argv[]) {
  cout << "Maximum sum of a subarray of size K: "
       << MaxSumSubArrayOfSizeK::findMaxSumSubArray(3, vector<int>{2, 1, 5, 1, 3, 2}) << endl;
  cout << "Maximum sum of a subarray of size K: "
       << MaxSumSubArrayOfSizeK::findMaxSumSubArray(2, vector<int>{2, 3, 4, 1, 5}) << endl;
}
```

Output 0.857s

```
Maximum sum of a subarray of size K: 9
Maximum sum of a subarray of size K: 7
```

The time complexity of the above algorithm will be $O(N * K)$, 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

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:

Java | Python3 | C++ | JS

```cpp
using namespace std;

#include <iostream>
#include <vector>

class MaxSumSubArrayOfSizeK {
 public:
  static int findMaxSumSubArray(int k, const vector<int>& arr) {
    int windowSum = 0, maxSum = 0;
    int windowStart = 0;
    for (int windowEnd = 0; windowEnd < arr.size(); 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 = max(maxSum, windowSum);
        windowSum -= arr[windowStart];  // subtract the element going out
        windowStart++;                  // slide the window ahead
      }
    }

    return maxSum;
  }
};

int main(int argc, char* argv[]) {
  cout << "Maximum sum of a subarray of size K: "
       << MaxSumSubArrayOfSizeK::findMaxSumSubArray(3, vector<int>{2, 1, 5, 1, 3, 2}) << endl;
  cout << "Maximum sum of a subarray of size K: "
       << MaxSumSubArrayOfSizeK::findMaxSumSubArray(2, vector<int>{2, 3, 4, 1, 5}) << endl;
}
```

Output 1.338s

```
Maximum sum of a subarray of size K: 9
Maximum sum of a subarray of size K: 7
```

Time Complexity

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

Space Complexity

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

Smallest Subarray with a given sum (easy)

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 great 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:

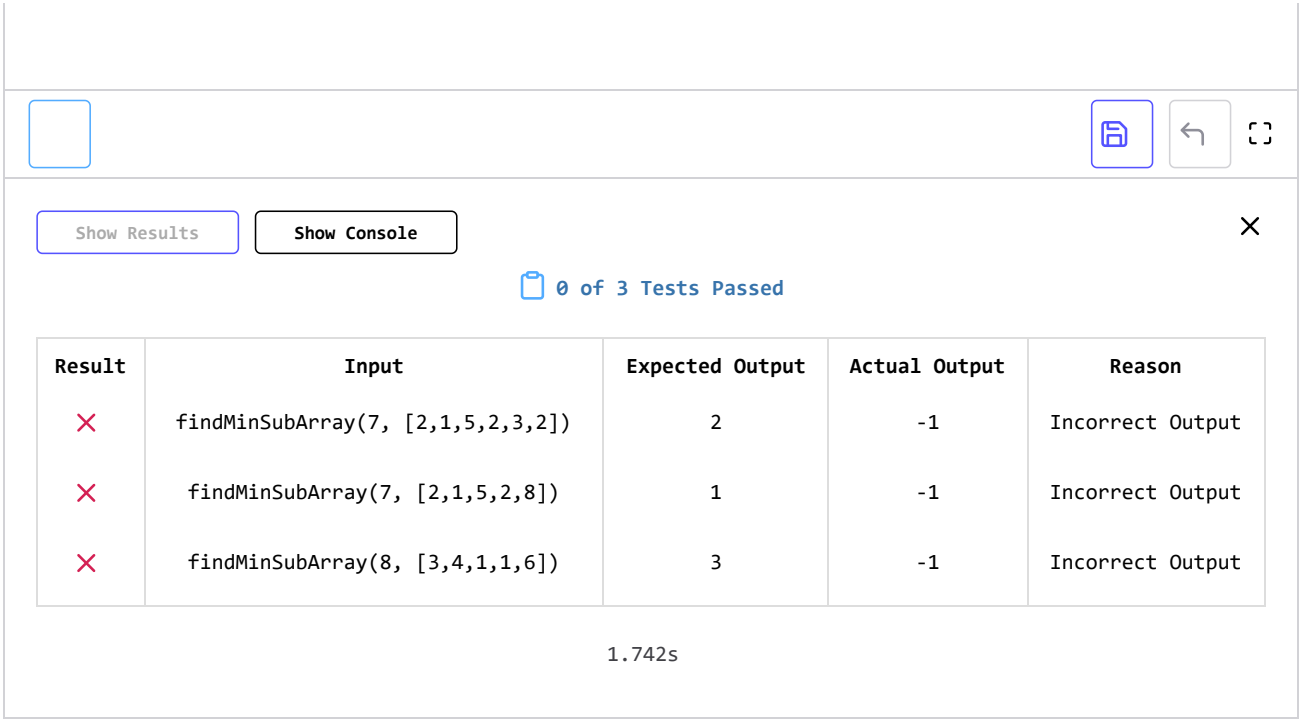
```
Input: [3, 4, 1, 1, 6], S=8
Output: 3
Explanation: Smallest subarrays with a sum greater than or equal to '8' are [3, 4, 1] o
r [1, 1, 6].
```

Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

```
using namespace std;

#include <iostream>
#include <limits>
#include <vector>

class MinSizeSubArraySum {
 public:
  static int findMinSubArray(int S, const vector<int>& arr) {
    // TODO: Write your code here
    return -1;
  }
};

```

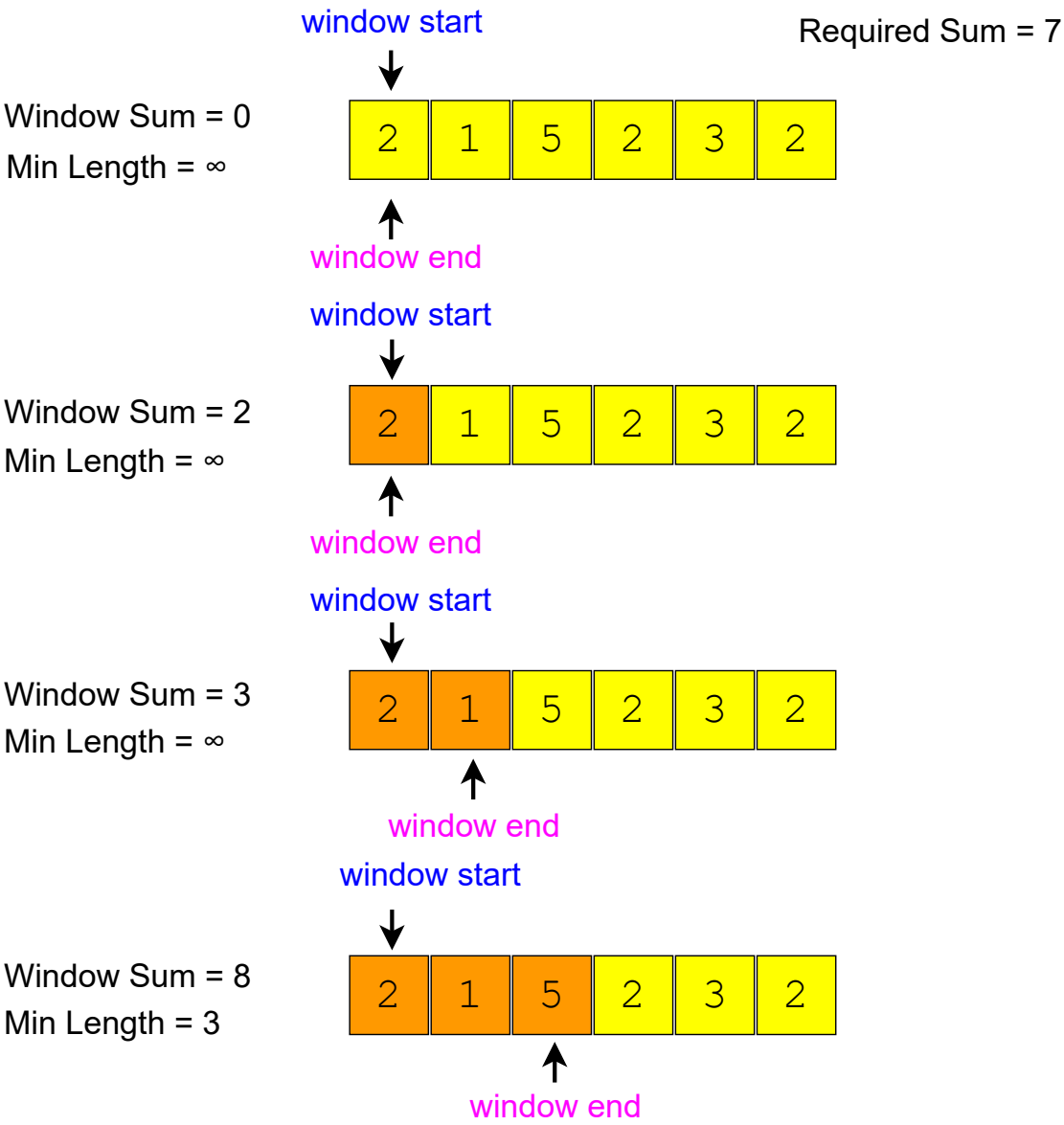


Solution

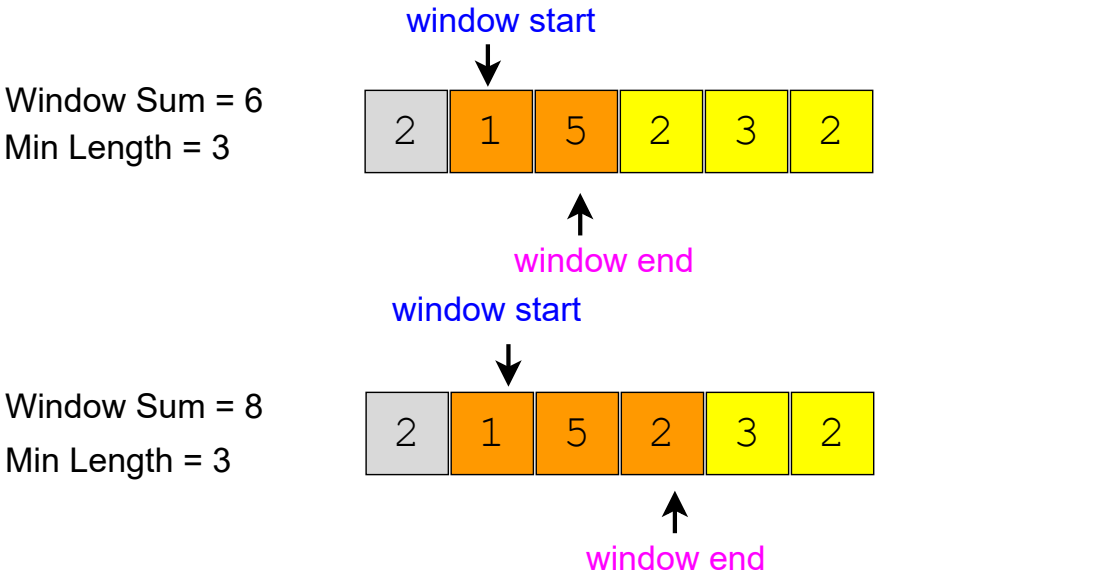
This problem follows the **Sliding Window** pattern and we can use a similar strategy as discussed in Maximum Sum Subarray of Size K. There is one difference though: in this problem, the size of the sliding window is not fixed. Here is how we will solve this problem:

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.

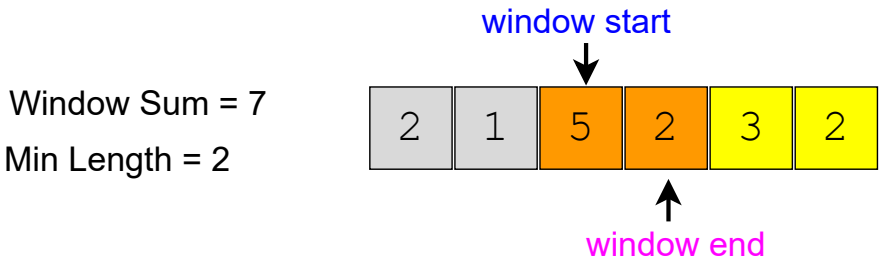
Here is the visual representation of this algorithm for the Example-1



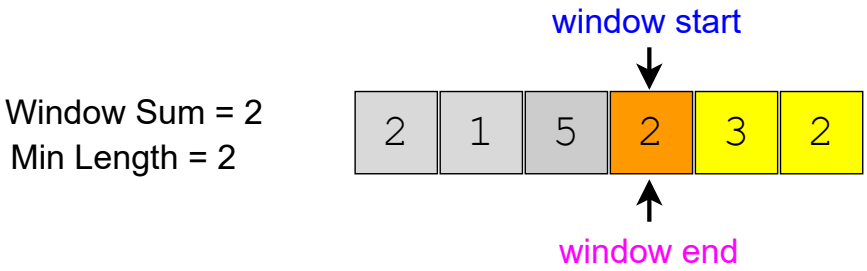
Window Sum >= 7, let's shrink the sliding window

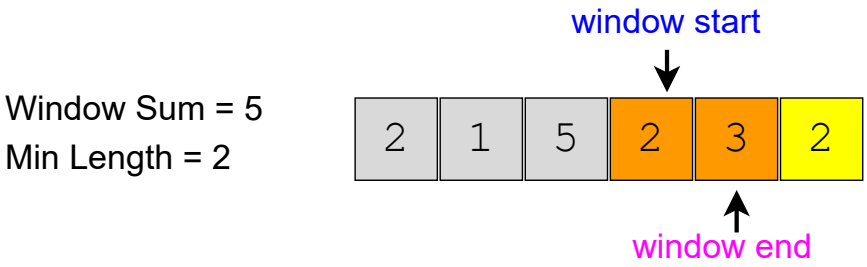


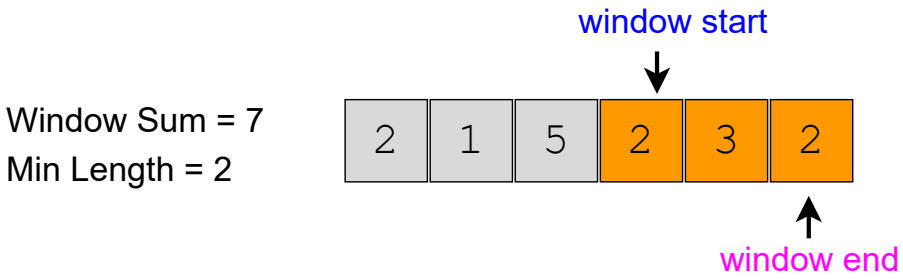
Window Sum >= 7, let's shrink the sliding window



Window Sum still >= 7, let's shrink the sliding window







Code

Here is what our algorithm will look:

Java | Python3 | C++ | JS

```cpp
using namespace std;

#include <iostream>
#include <limits>
#include <vector>

class MinSizeSubArraySum {
 public:
  static int findMinSubArray(int S, const vector<int>& arr) {
    int windowSum = 0, minLength = numeric_limits<int>::max();
    int windowStart = 0;
    for (int windowEnd = 0; windowEnd < arr.size(); 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 = min(minLength, windowEnd - windowStart + 1);
        windowSum -= arr[windowStart];  // subtract the element going out
        windowStart++;                  // slide the window ahead
      }
    }

    return minLength == numeric_limits<int>::max() ? 0 : minLength;
  }
};

int main(int argc, char* argv[]) {
  int result = MinSizeSubArraySum::findMinSubArray(7, vector<int>{2, 1, 5, 2, 3, 2});
  cout << "Smallest subarray length: " << result << endl;
  result = MinSizeSubArraySum::findMinSubArray(7, vector<int>{2, 1, 5, 2, 8});
  cout << "Smallest subarray length: " << result << endl;
  result = MinSizeSubArraySum::findMinSubArray(8, vector<int>{3, 4, 1, 1, 6});
  cout << "Smallest subarray length: " << result << endl;
}

```

Output 1.085s

```
Smallest subarray length: 2
Smallest subarray length: 1
Smallest subarray length: 3
```

Time Complexity

The time complexity of the above algorithm will be $O(N)$. 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 $O(N + N)$ which is asymptotically equivalent to $O(N)$.

Space Complexity

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

Longest Substring with K Distinct Characters (medium)

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 "cbbe
b" & "bbebi".
```

Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

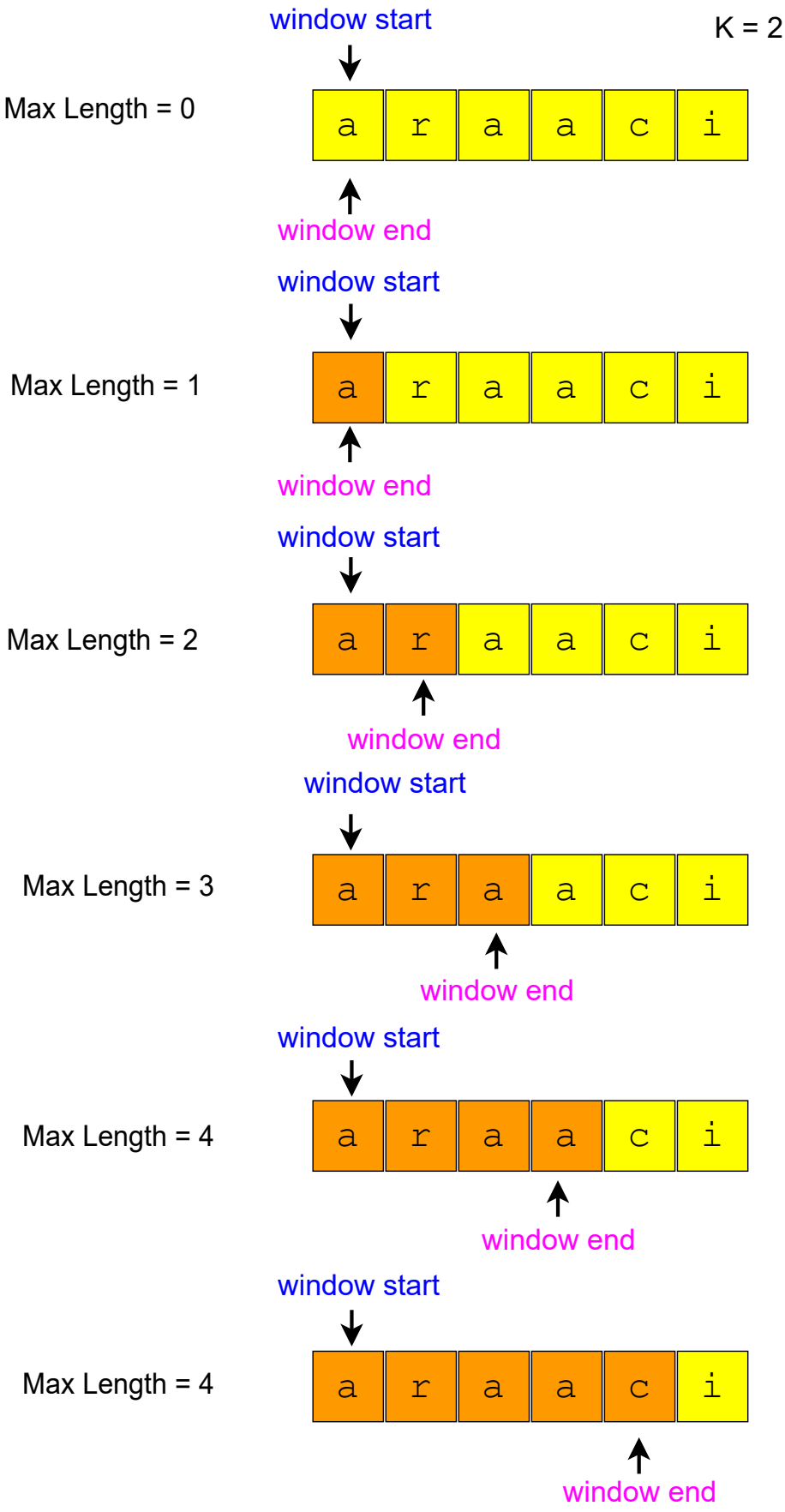
```
1  using namespace std;
2
3  #include <iostream>
4  #include <string>
5  #include <unordered_map>
6
7  class LongestSubstringKDistinct {
8   public:
9    static int findLength(const string& str, int k) {
10     int maxLength = 0;
11     // TODO: Write your code here
12     return maxLength;
13   }
14 };
15
```

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 given sum. We can use a **HashMap** to remember the frequency of each character we have processed. Here is how we will solve this problem:

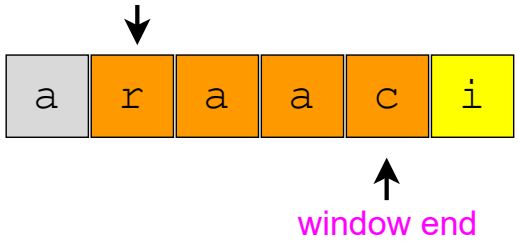
1. First, we will insert characters from the beginning of the string until we have 'K' distinct characters in the **HashMap**.
2. These characters will constitute our sliding window. We are asked to find the longest such window having no more than 'K' distinct characters. We will remember the length 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 frequency of the character 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:



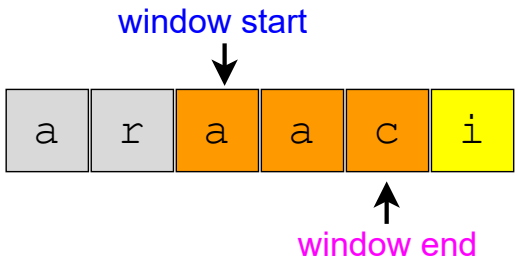
Number of distinct characters > 2, let's shrink the sliding window

window start

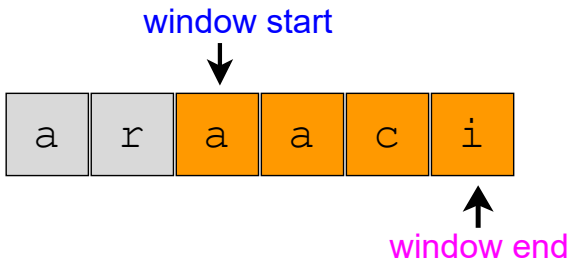


Number of distinct characters are still > 2, let's shrink the sliding window

Max Length = 4

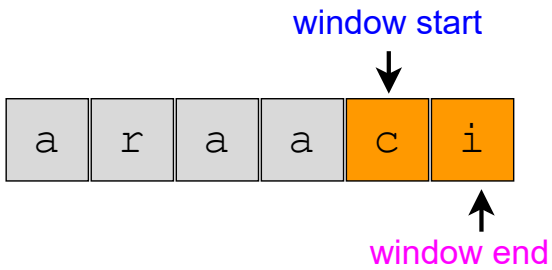


Max Length = 4



Number of distinct character > 2, let's shrink the sliding window

Max Length = 4



Code

Here is how our algorithm will look:

Java | Python3 | C++ | JS

```cpp
using namespace std;

#include <iostream>
#include <string>
#include <unordered_map>

class LongestSubstringKDistinct {
public:
  static int findLength(const string &str, int k) {
    int windowStart = 0, maxLength = 0;
    unordered_map<char, int> charFrequencyMap;
    // in the following loop we'll try to extend the range [windowStart, windowEnd]
    for (int windowEnd = 0; windowEnd < str.length(); windowEnd++) {
      char rightChar = str[windowEnd];
      charFrequencyMap[rightChar]++;
      // shrink the sliding window, until we are left with 'k' distinct characters in the frequency
      // map
      while ((int)charFrequencyMap.size() > k) {
        char leftChar = str[windowStart];
        charFrequencyMap[leftChar]--;
        if (charFrequencyMap[leftChar] == 0) {
          charFrequencyMap.erase(leftChar);
        }
        windowStart++; // shrink the window
      }
      maxLength = max(maxLength, windowEnd - windowStart + 1); // remember the maximum length so far
    }

    return maxLength;
  }
};

int main(int argc, char *argv[]) {
  cout << "Length of the longest substring: " << LongestSubstringKDistinct::findLength("araaci", 2)
       << endl;
  cout << "Length of the longest substring: " << LongestSubstringKDistinct::findLength("araaci", 1)
       << endl;
  cout << "Length of the longest substring: " << LongestSubstringKDistinct::findLength("cbbebi", 3)
       << endl;
}

```

Output 0.730s

```
Length of the longest substring: 4
Length of the longest substring: 2
Length of the longest substring: 5
```

Time Complexity

The time complexity of the above algorithm 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

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

Fruits into Baskets (medium)

Problem Statement

Given an array of characters where each character represents a fruit tree, you are given **two baskets** and your goal is to put **maximum number of fruits in each basket**. The only restriction is that **each basket can have only one type of fruit**.

You can start with any tree, but once you have started you can't skip a tree. You will pick one fruit from each 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 the 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 subarra
y ['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']
```

Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

```cpp
using namespace std;

#include <iostream>
#include <unordered_map>
#include <vector>

class MaxFruitCountOf2Types {
 public:
  static int findLength(const vector<char>& arr) {
    int maxLength = 0;
    // TODO: Write your code here
    return maxLength;
  }
};
```

Show Results | Show Console

0 of 2 Tests Passed

Result	Input	Expected Output	Actual Output	Reason
✗	fruits_into_baskets([A, B, C, A, C])	3	-1	Incorrect Output
✗	fruits_into_baskets([A, B, C, B, B, C])	5	-1	Incorrect Output

0.468s

Solution

This problem follows the **Sliding Window** pattern and is quite similar to Longest Substring with K Distinct Characters. 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

Here is what our algorithm will look like, only the highlighted lines are different from Longest Substring with K Distinct Characters:

Java | Python3 | C++ | JS

```cpp
using namespace std;

#include <iostream>
#include <unordered_map>
#include <vector>

class MaxFruitCountOf2Types {
 public:
  static int findLength(const vector<char>& arr) {
    int windowStart = 0, maxLength = 0;
    unordered_map<char, int> fruitFrequencyMap;
    // try to extend the range [windowStart, windowEnd]
    for (int windowEnd = 0; windowEnd < arr.size(); windowEnd++) {
      fruitFrequencyMap[arr[windowEnd]]++;
      // shrink the sliding window, until we are left with '2' fruits in the frequency map
      while ((int)fruitFrequencyMap.size() > 2) {
        fruitFrequencyMap[arr[windowStart]]--;
        if (fruitFrequencyMap[arr[windowStart]] == 0) {
          fruitFrequencyMap.erase(arr[windowStart]);
        }
        windowStart++;  // shrink the window
      }
      maxLength = max(maxLength, windowEnd - windowStart + 1);
    }

    return maxLength;
  }
};

int main(int argc, char* argv[]) {
  cout << "Maximum number of fruits: "
       << MaxFruitCountOf2Types::findLength(vector<char>{'A', 'B', 'C', 'A', 'C'}) << endl;
  cout << "Maximum number of fruits: "
       << MaxFruitCountOf2Types::findLength(vector<char>{'A', 'B', 'C', 'B', 'B', 'C'}) << endl;
}
```

Output 1.222s

```
Maximum number of fruits: 3
Maximum number of fruits: 5
```

Time Complexity

The time complexity of the above algorithm 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

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

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.

No-repeat Substring (hard)

Problem Statement

Given a string, find the **length of the longest substring** which has **no repeating characters**.

Example 1:

```
Input: String="aabccbb"
Output: 3
Explanation: The longest substring without any repeating characters is "abc".
```

Example 2:

```
Input: String="abbbb"
Output: 2
Explanation: The longest substring without any repeating characters is "ab".
```

Example 3:

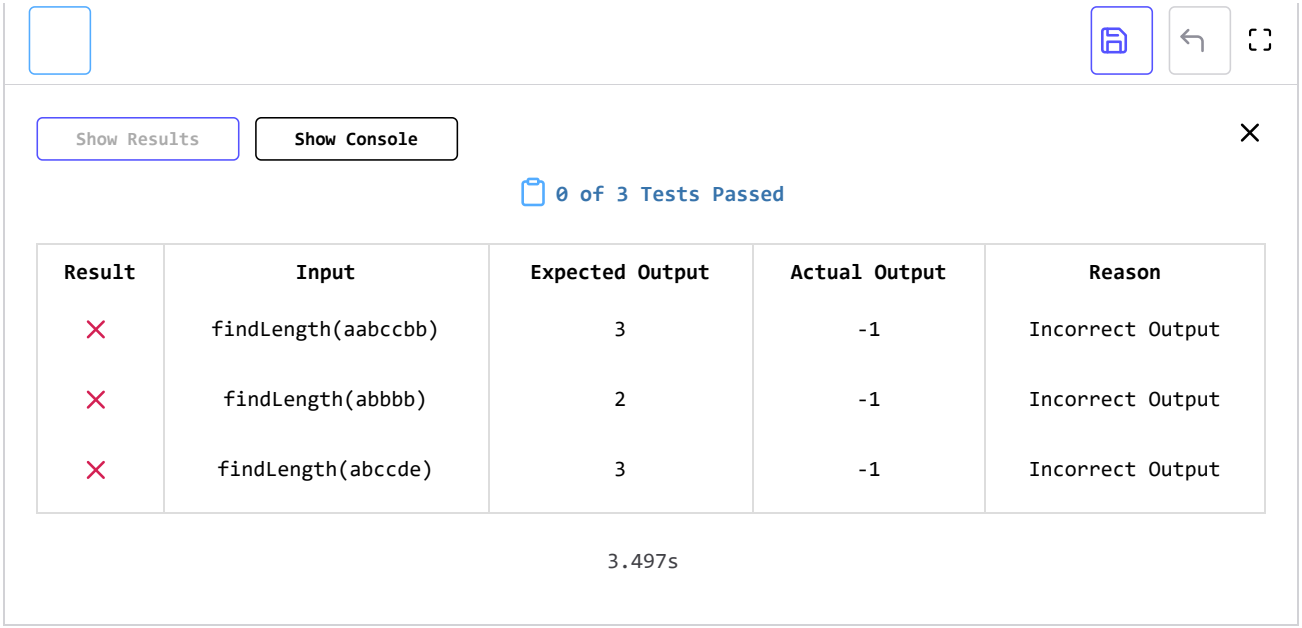
```
Input: String="abccde"
Output: 3
Explanation: Longest substrings without any repeating characters are "abc" & "cde".
```

Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

```
using namespace std;

#include <iostream>
#include <string>
#include <unordered_map>

class NoRepeatSubstring {
 public:
  static int findLength(const string& str) {
    int maxLength = 0;
    // TODO: Write your code here
    return maxLength;
  }
};

```



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. We can use a **HashMap** to remember the last index of each character we have processed. Whenever we get a repeating character we will shrink our sliding window to ensure that we always have distinct characters in the sliding window.

Code

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```cpp
using namespace std;

#include <iostream>
#include <string>
#include <unordered_map>

class NoRepeatSubstring {
 public:
  static int findLength(const string& str) {
    int windowStart = 0, maxLength = 0;
    unordered_map<char, int> charIndexMap;
    // try to extend the range [windowStart, windowEnd]
    for (int windowEnd = 0; windowEnd < str.length(); windowEnd++) {
      char rightChar = str[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.find(rightChar) != charIndexMap.end()) {
        // 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 = max(windowStart, charIndexMap[rightChar] + 1);
      }
      charIndexMap[rightChar] = windowEnd;  // insert the 'rightChar' into the map
      maxLength =
          max(maxLength, windowEnd - windowStart + 1);  // remember the maximum length so far
    }

    return maxLength;
  }
};

int main(int argc, char* argv[]) {
  cout << "Length of the longest substring: " << NoRepeatSubstring::findLength("aabccbb") << endl;
  cout << "Length of the longest substring: " << NoRepeatSubstring::findLength("abbbb") << endl;
  cout << "Length of the longest substring: " << NoRepeatSubstring::findLength("abccde") << endl;
}

```

Output 0.955s

```
Length of the longest substring: 3
Length of the longest substring: 2
Length of the longest substring: 3
```

Time Complexity

The time complexity of the above algorithm will be $O(N)$ where 'N' is the number of characters in the input string.

Space Complexity

The space complexity of the algorithm 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 repeating 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**.

Longest Substring with Same Letters after Replacement (hard)

Problem Statement

Given a string with lowercase letters only, if you are allowed to **replace no more than 'k' letters** with any letter, find the **length of the longest substring having the same letters** after replacement.

Example 1:

```
Input: String="aabccbb", k=2
Output: 5
Explanation: Replace the two 'c' with 'b' to have a longest repeating substring "bbbbb".
```

Example 2:

```
Input: String="abbcb", k=1
Output: 4
Explanation: Replace the 'c' with 'b' to have a longest repeating substring "bbbb".
```

Example 3:

```
Input: String="abccde", k=1
Output: 3
Explanation: Replace the 'b' or 'd' with 'c' to have the longest repeating substring "ccc".
```

Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

```
1  using namespace std;
2
3  #include <iostream>
4  #include <string>
5  #include <unordered_map>
6
7  class CharacterReplacement {
8   public:
9    static int findLength(const string& str, int k) {
10     int maxLength = 0;
11     // TODO: Write your code here
12     return maxLength;
13   }
14 };
15
```

Solution

This problem follows the **Sliding Window** pattern and we can use a similar dynamic sliding window strategy as discussed in No-repeat Substring. We can use a HashMap to count the frequency of each letter.

We'll iterate through the string to add one letter at a time in the window. We'll also keep track of the count of the maximum repeating letter in any window (let's call it `maxRepeatLetterCount`). So at any time, we know that we can have a window which has one letter repeating `maxRepeatLetterCount` times, this means we should try to replace the remaining letters. If we have more than 'k' remaining letters, we should shrink the window as we are not allowed to replace more than 'k' letters.

Code

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```cpp
using namespace std;

#include <iostream>
#include <string>
#include <unordered_map>

class CharacterReplacement {
public:
  static int findLength(const string &str, int k) {
    int windowStart = 0, maxLength = 0, maxRepeatLetterCount = 0;
    unordered_map<char, int> letterFrequencyMap;
    // try to extend the range [windowStart, windowEnd]
    for (int windowEnd = 0; windowEnd < str.length(); windowEnd++) {
      char rightChar = str[windowEnd];
      letterFrequencyMap[rightChar]++;
      maxRepeatLetterCount = max(maxRepeatLetterCount, letterFrequencyMap[rightChar]);

      // current window size is from windowStart to windowEnd, overall we have a letter which is
      // repeating 'maxRepeatLetterCount' times, this means we can have a window which has one
      // letter repeating 'maxRepeatLetterCount' times and the remaining letters we should replace.
      // if the remaining letters are more than 'k', it is the time to shrink the window as we
      // are not allowed to replace more than 'k' letters
      if (windowEnd - windowStart + 1 - maxRepeatLetterCount > k) {
        char leftChar = str[windowStart];
        letterFrequencyMap[leftChar]--;
        windowStart++;
      }

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

    return maxLength;
  }
};

int main(int argc, char *argv[]) {
  cout << CharacterReplacement::findLength("aabccbb", 2) << endl;
  cout << CharacterReplacement::findLength("abbcb", 1) << endl;
  cout << CharacterReplacement::findLength("abccde", 1) << endl;
}
```

Output 2.191s

```
5
4
3
```

Time Complexity

The time complexity of the above algorithm will be $O(N)$ where 'N' is the number of letters in the input string.

Space Complexity

As we are expecting only the lower case letters in the input string, we can conclude that the space complexity will be $O(26)$, to store each letter's frequency in the **HashMap**, which is asymptotically equal to $O(1)$.

Longest Subarray with Ones after Replacement (hard)

Problem Statement

Given an array containing 0s and 1s, if you are allowed to **replace no more than 'k' 0s with 1s**, find the length of the **longest contiguous subarray having all 1s**.

Example 1:

```
Input: Array=[0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1], k=2
Output: 6
Explanation: Replace the '0' at index 5 and 8 to have the longest contiguous subarray of 1
s having length 6.
```

Example 2:

```
Input: Array=[0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1], k=3
Output: 9
Explanation: Replace the '0' at index 6, 9, and 10 to have the longest contiguous subarra
y of 1s having length 9.
```

Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

```
1  using namespace std;
2
3  #include <iostream>
4  #include <vector>
5
6  class ReplacingOnes {
7   public:
8    static int findLength(const vector<int>& arr, int k) {
9      int  maxLength = 0;
10     // TODO: Write your code here
11     return maxLength;
12   }
13 };
```

Show Results | Show Console

0 of 2 Tests Passed

Result	Input	Expected Output	Actual Output	Reason
✗	findLength([0, 1, 1, 0, 0, 0, 1, 1, 0, 1 …	6	-1	Incorrect Output
✗	findLength([0, 1, 0, 0, 1, 1, 0, 1, 1, 0 …	9	-1	Incorrect Output

3.860s

Solution

This problem follows the **Sliding Window** pattern and is quite similar to Longest Substring with same Letters after Replacement. The only difference is that, in the problem, we only have two characters (1s and 0s) in the input arrays.

Following a similar approach, we'll iterate through the array to add one number at a time in the window. We'll also keep track of the maximum number of repeating 1s in the current window (let's call it `maxOnesCount`). So at any time, we know that we can have a window which has 1s repeating `maxOnesCount` time, so we should try to replace the remaining 0s. If we have more than 'k' remaining 0s, we should shrink the window as we are not allowed to replace more than 'k' 0s.

Code

Here is how our algorithm will look like:

Java | Python3 | C++ | JS

```cpp
using namespace std;

#include <iostream>
#include <vector>

class ReplacingOnes {
 public:
  static int findLength(const vector<int>& arr, int k) {
    int windowStart = 0, maxLength = 0, maxOnesCount = 0;
    // try to extend the range [windowStart, windowEnd]
    for (int windowEnd = 0; windowEnd < arr.size(); windowEnd++) {
      if (arr[windowEnd] == 1) {
        maxOnesCount++;
      }

      // current window size is from windowStart to windowEnd, overall we have a maximum of 1s
      // repeating a maximum of 'maxOnesCount' times, this means that we can have a window with
      // 'maxOnesCount' 1s and the remaining are 0s which should replace with 1s.
      // now, if the remaining 0s are more than 'k', it is the time to shrink the window as we
      // are not allowed to replace more than 'k' Os
      if (windowEnd - windowStart + 1 - maxOnesCount > k) {
        if (arr[windowStart] == 1) {
          maxOnesCount--;
        }
        windowStart++;
      }

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

    return maxLength;
  }
};

int main(int argc, char* argv[]) {
  cout << ReplacingOnes::findLength(vector<int>{0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1}, 2) << endl;
  cout << ReplacingOnes::findLength(vector<int>{0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1}, 3) << endl;
}
```

Output 1.743s

```
6
9
```

Time Complexity

The time complexity of the above algorithm will be $O(N)$ where 'N' is the count of numbers in the input array.

Space Complexity

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

Problem Challenge 1

Permutation in a String (hard)

Given a string and a pattern, find out if the **string contains any permutation of the pattern**.

Permutation is defined as the re-arranging of the characters of the string. For example, “abc” has the following six permutations:

1. abc
2. acb
3. bac
4. bca
5. cab
6. cba

If a string has ‘n’ distinct characters it will have $n!$ permutations.

Example 1:

```
Input: String="oidbcaf", Pattern="abc"
Output: true
Explanation: The string contains "bca" which is a permutation of the given pattern.
```

Example 2:

```
Input: String="odicf", Pattern="dc"
Output: false
Explanation: No permutation of the pattern is present in the given string as a substring.
```

Example 3:

```
Input: String="bcdxabcdy", Pattern="bcdyabcdx"
Output: true
Explanation: Both the string and the pattern are a permutation of each other.
```

Example 4:

```
Input: String="aaacb", Pattern="abc"
Output: true
Explanation: The string contains "acb" which is a permutation of the given pattern.
```

Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

```cpp
using namespace std;

#include <iostream>
#include <string>

class StringPermutation {
 public:
  static bool findPermutation(const string &str, const string &pattern) {
    // TODO: Write your code here
    return false;
  }
};

```

Show Results | Show Console

1 of 4 Tests Passed

Result	Input	Expected Output	Actual Output	Reason
✗	find_permutation(oidbcaf, abc)	True	False	Incorrect Output
✗	find_permutation(bcdxabcdy, bcdyabcdx)	True	False	Incorrect Output
✗	find_permutation(aaacb, abc)	True	False	Incorrect Output
✓	find_permutation(odicf, dc)	False	False	Succeeded

1.217s

Solution Review: Problem Challenge 1

Permutation in a String (hard)

Given a string and a pattern, find out if the **string contains any permutation of the pattern**.

Permutation is defined as the re-arranging of the characters of the string. For example, “abc” has the following six permutations:

1. abc
2. acb
3. bac
4. bca
5. cab
6. cba

If a string has ‘n’ distinct characters it will have $n!$ permutations.

Example 1:

```
Input: String="oidbcaf", Pattern="abc"
Output: true
Explanation: The string contains "bca" which is a permutation of the given pattern.
```

Example 2:

```
Input: String="odicf", Pattern="dc"
Output: false
Explanation: No permutation of the pattern is present in the given string as a substring.
```

Example 3:

```
Input: String="bcdxabcdy", Pattern="bcdyabcdx"
Output: true
Explanation: Both the string and the pattern are a permutation of each other.
```

Example 4:

```
Input: String="aaacb", Pattern="abc"
Output: true
Explanation: The string contains "acb" which is a permutation of the given pattern.
```

Solution

This problem follows the **Sliding Window** pattern and we can use a similar sliding window strategy as discussed in Longest Substring with K Distinct Characters. We can use a **HashMap** to remember the frequencies of all characters in the given pattern. Our goal will be to match all the characters from this **HashMap** with a sliding window in the given string. Here are the steps of our algorithm:

1. Create a **HashMap** to calculate the frequencies of all characters in the pattern.
2. Iterate through the string, adding one character at a time in the sliding window.
3. If the character being added matches a character in the **HashMap**, decrement its frequency in the map. If the character frequency becomes zero, we got a complete match.
4. If at any time, the number of characters matched is equal to the number of distinct characters in the pattern (i.e., total characters in the **HashMap**), we have gotten our required permutation.
5. If the window size is greater than the length of the pattern, shrink the window to make it equal to the size of the pattern. At the same time, if the character going out was part of the pattern, put it back in the frequency **HashMap**.

Code

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```cpp
using namespace std;

#include <iostream>
#include <string>
#include <unordered_map>

class StringPermutation {
 public:
  static bool findPermutation(const string &str, const string &pattern) {
    int windowStart = 0, matched = 0;
    unordered_map<char, int> charFrequencyMap;
    for (auto chr : pattern) {
      charFrequencyMap[chr]++;
    }

    // our goal is to match all the characters from the 'charFrequencyMap' with the current window
    // try to extend the range [windowStart, windowEnd]
    for (int windowEnd = 0; windowEnd < str.length(); windowEnd++) {
      char rightChar = str[windowEnd];
      if (charFrequencyMap.find(rightChar) != charFrequencyMap.end()) {
        // decrement the frequency of the matched character
        charFrequencyMap[rightChar]--;
        if (charFrequencyMap[rightChar] == 0) {  // character is completely matched
          matched++;
        }
      }

      if (matched == (int)charFrequencyMap.size()) {
        return true;
      }

      if (windowEnd >= pattern.length() - 1) {  // shrink the window
        char leftChar = str[windowStart++];
        if (charFrequencyMap.find(leftChar) != charFrequencyMap.end()) {
          if (charFrequencyMap[leftChar] == 0) {
            matched--;  // before putting the character back, decrement the matched count
          }
          // put the character back for matching
          charFrequencyMap[leftChar]++;
        }
      }
    }

    return false;
  }
};

int main(int argc, char *argv[]) {
  cout << "Permutation exist: " << StringPermutation::findPermutation("oidbcaf", "abc") << endl;
  cout << "Permutation exist: " << StringPermutation::findPermutation("odicf", "dc") << endl;
  cout << "Permutation exist: " << StringPermutation::findPermutation("bcdxabcdy", "bcdyabcdx") << endl;
  cout << "Permutation exist: " << StringPermutation::findPermutation("aaacb", "abc") << endl;
}

```

Output 1.223s

```
Permutation exist: 1
Permutation exist: 0
```

```
Permutation exist: 1
Permutation exist: 1
```

Time Complexity

The time complexity of the above algorithm will be $O(N + M)$ where 'N' and 'M' are the number of characters in the input string and the pattern respectively.

Space Complexity

The space complexity of the algorithm is $O(M)$ since in the worst case, the whole pattern can have distinct characters which will go into the **HashMap**.

Problem Challenge 2

String Anagrams (hard)

Given a string and a pattern, find **all anagrams of the pattern in the given string**.

Anagram is actually a **Permutation** of a string. For example, “abc” has the following six anagrams:

1. abc
2. acb
3. bac
4. bca
5. cab
6. cba

Write a function to return a list of starting indices of the anagrams of the pattern in the given string.

Example 1:

```
Input: String="ppqp", Pattern="pq"
Output: [1, 2]
Explanation: The two anagrams of the pattern in the given string are "pq" and "qp".
```

Example 2:

```
Input: String="abbcabc", Pattern="abc"
Output: [2, 3, 4]
Explanation: The three anagrams of the pattern in the given string are "bca", "cab", and "a
bc".
```

Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

```cpp
using namespace std;

#include <iostream>
#include <string>
#include <unordered_map>
#include <vector>

class StringAnagrams {
 public:
  static vector<int> findStringAnagrams(const string &str, const string &pattern) {
    vector<int> resultIndices;
    // TODO: Write your code here
    return resultIndices;
  }
};

```

Show Results | Show Console

0 of 2 Tests Passed

Result	Input	Expected Output	Actual Output	Reason
✗	findStringAnagrams(ppqp, pq)	[1, 2]	[]	Incorrect Output
✗	findStringAnagrams(abbcabc, abc)	[2, 3, 4]	[]	Incorrect Output

3.888s

Solution Review: Problem Challenge 2

String Anagrams (hard)

Given a string and a pattern, find **all anagrams of the pattern in the given string**.

Anagram is actually a **Permutation** of a string. For example, “abc” has the following six anagrams:

1. abc
2. acb
3. bac
4. bca
5. cab
6. cba

Write a function to return a list of starting indices of the anagrams of the pattern in the given string.

Example 1:

```
Input: String="ppqp", Pattern="pq"
Output: [1, 2]
Explanation: The two anagrams of the pattern in the given string are "pq" and "qp".
```

Example 2:

```
Input: String="abbcabc", Pattern="abc"
Output: [2, 3, 4]
Explanation: The three anagrams of the pattern in the given string are "bca", "cab", and "a
bc".
```

Solution

This problem follows the **Sliding Window** pattern and is very similar to Permutation in a String. In this problem, we need to find every occurrence of any permutation of the pattern in the string. We will use a list to store the starting indices of the anagrams of the pattern in the string.

Code

Here is what our algorithm will look like, only the highlighted lines have changed from Permutation in a String:

```
using namespace std;

#include <iostream>
#include <string>
#include <unordered_map>
#include <vector>

class StringAnagrams {
 public:
  static vector<int> findStringAnagrams(const string &str, const string &pattern) {
    int windowStart = 0, matched = 0;
    unordered_map<char, int> charFrequencyMap;
    for (auto chr : pattern) {
      charFrequencyMap[chr]++;
    }

    vector<int> resultIndices;
    // our goal is to match all the characters from the map with the current window
    for (int windowEnd = 0; windowEnd < str.length(); windowEnd++) {
      char rightChar = str[windowEnd];
      // decrement the frequency of the matched character
      if (charFrequencyMap.find(rightChar) != charFrequencyMap.end()) {
        charFrequencyMap[rightChar]--;
        if (charFrequencyMap[rightChar] == 0) {
          matched++;
        }
      }

      if (matched == (int)charFrequencyMap.size()) {  // have we found an anagram?
        resultIndices.push_back(windowStart);
      }

      if (windowEnd >= pattern.length() - 1) {  // shrink the window
        char leftChar = str[windowStart++];
        if (charFrequencyMap.find(leftChar) != charFrequencyMap.end()) {
          if (charFrequencyMap[leftChar] == 0) {
            matched--;  // before putting the character back, decrement the matched count
          }
          // put the character back
          charFrequencyMap[leftChar]++;
        }
      }
    }

    return resultIndices;
  }
};

int main(int argc, char *argv[]) {
  auto result = StringAnagrams::findStringAnagrams("ppqp", "pq");
  for (auto num : result) {
    cout << num << " ";
  }
  cout << endl;

  result = StringAnagrams::findStringAnagrams("abbcabc", "abc");
  for (auto num : result) {
    cout << num << " ";
```

```
62
```

Output 1.327s

```
1 2
2 3 4
```

Time Complexity

The time complexity of the above algorithm will be $O(N + M)$ where 'N' and 'M' are the number of characters in the input string and the pattern respectively.

Space Complexity

The space complexity of the algorithm is $O(M)$ since in the worst case, the whole pattern can have distinct characters which will go into the **HashMap**. In the worst case, we also need $O(N)$ space for the result list, this will happen when the pattern has only one character and the string contains only that character.

Problem Challenge 3

Smallest Window containing Substring (hard)

Given a string and a pattern, find the **smallest substring** in the given string which has **all the characters of the given pattern**.

Example 1:

```
Input: String="aabdec", Pattern="abc"
Output: "abdec"
Explanation: The smallest substring having all characters of the pat
tern is "abdec"
```

Example 2:

```
Input: String="abdabca", Pattern="abc"
Output: "abc"
Explanation: The smallest substring having all characters of the pat
tern is "abc".
```

Example 3:

```
Input: String="adcad", Pattern="abc"
Output: ""
Explanation: No substring in the given string has all characters o
f the pattern.
```

Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

```cpp
using namespace std;

#include <iostream>
#include <string>
#include <unordered_map>

class MinimumWindowSubstring {
 public:
  static string findSubstring(const string &str, const string &pattern) {
    // TODO: Write your code here
    return "";
  }
};

```

1 of 3 Tests Passed

Result	Input	Expected Output	Actual Output	Reason
✗	findSubstring(aabdec, abc)	abdec		Incorrect Output
✗	findSubstring(abdabca, abc)	abc		Incorrect Output
✓	findSubstring(adcad, abc)			Succeeded

3.392s

Solution Review: Problem Challenge 3

Smallest Window containing Substring (hard)

Given a string and a pattern, find the **smallest substring** in the given string which has **all the characters of the given pattern**.

Example 1:

```
Input: String="aabdec", Pattern="abc"
Output: "abdec"
Explanation: The smallest substring having all characters of the pattern is "abdec"
```

Example 2:

```
Input: String="abdabca", Pattern="abc"
Output: "abc"
Explanation: The smallest substring having all characters of the pattern is "abc".
```

Example 3:

```
Input: String="adcad", Pattern="abc"
Output: ""
Explanation: No substring in the given string has all characters of the pattern.
```

Solution

This problem follows the **Sliding Window** pattern and has a lot of similarities with Permutation in a String with one difference. In this problem, we need to find a substring having all characters of the pattern which means that the required substring can have some additional

characters and doesn't need to be a permutation of the pattern. Here is how we will manage these differences:

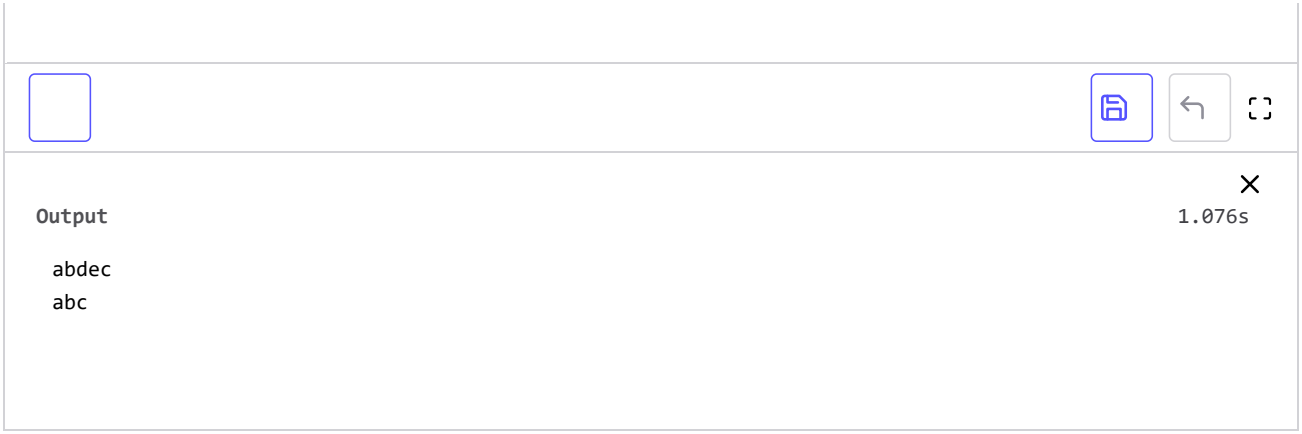
1. We will keep a running count of every matching instance of a character.
2. Whenever we have matched all the characters, we will try to shrink the window from the beginning, keeping track of the smallest substring that has all the matching characters.
3. We will stop the shrinking process as soon as we remove a matched character from the sliding window. One thing to note here is that we could have redundant matching characters, e.g., we might have two 'a' in the sliding window when we only need one 'a'. In that case, when we encounter the first 'a', we will simply shrink the window without decrementing the matched count. We will decrement the matched count when the second 'a' goes out of the window.

Code

Here is how our algorithm will look; only the highlighted lines have changed from Permutation in a String:

Java | Python3 | C++ | JS

```cpp
using namespace std;

#include <iostream>
#include <string>
#include <unordered_map>

class MinimumWindowSubstring {
 public:
  static string findSubstring(const string &str, const string &pattern) {
    int windowStart = 0, matched = 0, minLength = str.length() + 1, subStrStart = 0;
    unordered_map<char, int> charFrequencyMap;
    for (auto chr : pattern) {
      charFrequencyMap[chr]++;
    }

    // try to extend the range [windowStart, windowEnd]
    for (int windowEnd = 0; windowEnd < str.length(); windowEnd++) {
      char rightChar = str[windowEnd];
      if (charFrequencyMap.find(rightChar) != charFrequencyMap.end()) {
        charFrequencyMap[rightChar]--;
        if (charFrequencyMap[rightChar] >= 0) {  // count every matching of a character
          matched++;
        }
      }

      // shrink the window if we can, finish as soon as we remove a matched character
      while (matched == pattern.length()) {
        if (minLength > windowEnd - windowStart + 1) {
          minLength = windowEnd - windowStart + 1;
          subStrStart = windowStart;
        }

        char leftChar = str[windowStart++];
        if (charFrequencyMap.find(leftChar) != charFrequencyMap.end()) {
          // note that we could have redundant matching characters, therefore we'll decrement the
          // matched count only when a useful occurrence of a matched character is going out of the
          // window
          if (charFrequencyMap[leftChar] == 0) {
            matched--;
          }
          charFrequencyMap[leftChar]++;
        }
      }
    }

    return minLength > str.length() ? "" : str.substr(subStrStart, minLength);
  }
};

int main(int argc, char *argv[]) {
  cout << MinimumWindowSubstring::findSubstring("aabdec", "abc") << endl;
  cout << MinimumWindowSubstring::findSubstring("abdabca", "abc") << endl;
  cout << MinimumWindowSubstring::findSubstring("adcad", "abc") << endl;
}

```



Time Complexity

The time complexity of the above algorithm will be $O(N + M)$ where 'N' and 'M' are the number of characters in the input string and the pattern respectively.

Space Complexity

The space complexity of the algorithm is $O(M)$ since in the worst case, the whole pattern can have distinct characters which will go into the **HashMap**. In the worst case, we also need $O(N)$ space for the resulting substring, which will happen when the input string is a permutation of the pattern.

Problem Challenge 4

Words Concatenation (hard)

Given a string and a list of words, find all the starting indices of substrings in the given string that are a **concatenation of all the given words** exactly once **without any overlapping** of words. It is given that all words are of the same length.

Example 1:

```
Input: String="catfoxcat", Words=["cat", "fox"]
Output: [0, 3]
Explanation: The two substring containing both the words are "catfox" & "foxcat".
```

Example 2:

```
Input: String="catcatfoxfox", Words=["cat", "fox"]
Output: [3]
Explanation: The only substring containing both the words is "catfox".
```

Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

```
using namespace std;

#include <iostream>
#include <string>
#include <unordered_map>
#include <vector>

class WordConcatenation {
 public:
  static vector<int> findWordConcatenation(const string &str, const vector<string> &words) {
    vector<int> resultIndices;
    // TODO: Write your code here
    return resultIndices;
  }
};

```

Solution Review: Problem Challenge 4

Words Concatenation (hard)

Given a string and a list of words, find all the starting indices of substrings in the given string that are a **concatenation of all the given words** exactly once **without any overlapping** of words. It is given that all words are of the same length.

Example 1:

```
Input: String="catfoxcat", Words=["cat", "fox"]
Output: [0, 3]
Explanation: The two substring containing both the words are "catfox" & "foxcat".
```

Example 2:

```
Input: String="catcatfoxfox", Words=["cat", "fox"]
Output: [3]
Explanation: The only substring containing both the words is "catfox".
```

Solution

This problem follows the **Sliding Window** pattern and has a lot of similarities with Maximum Sum Subarray of Size K. We will keep track of all the words in a **HashMap** and try to match them in the given string. Here are the set of steps for our algorithm:

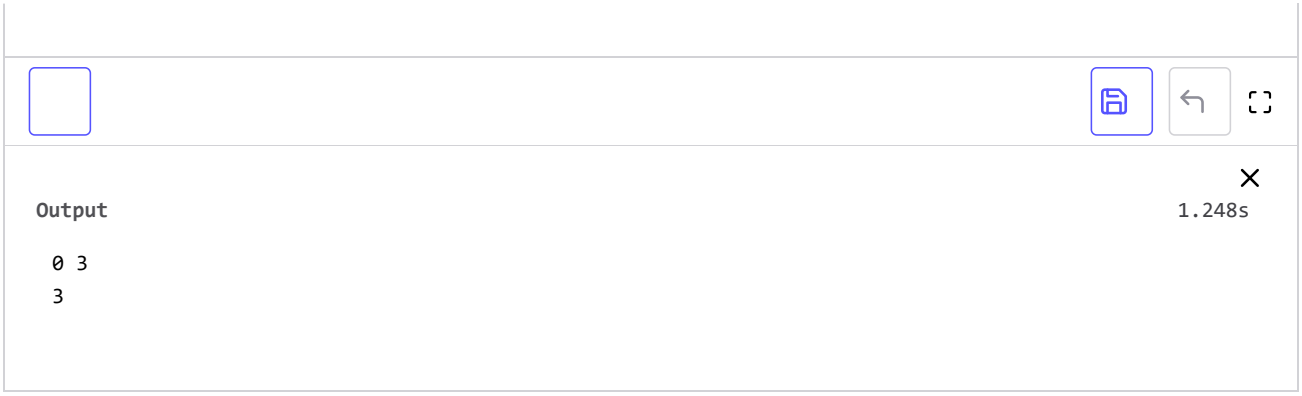
1. Keep the frequency of every word in a **HashMap**.
2. Starting from every index in the string, try to match all the words.
3. In each iteration, keep track of all the words that we have already seen in another **HashMap**.
4. If a word is not found or has a higher frequency than required, we can move on to the next character in the string.
5. Store the index if we have found all the words.

Code

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```cpp
using namespace std;

#include <iostream>
#include <string>
#include <unordered_map>
#include <vector>

class WordConcatenation {
 public:
  static vector<int> findWordConcatenation(const string &str, const vector<string> &words) {
    unordered_map<string, int> wordFrequencyMap;
    for (auto word : words) {
      wordFrequencyMap[word]++;
    }

    vector<int> resultIndices;
    int wordsCount = words.size(), wordLength = words[0].length();

    for (int i = 0; i <= str.length() - wordsCount * wordLength; i++) {
      unordered_map<string, int> wordsSeen;
      for (int j = 0; j < wordsCount; j++) {
        int nextWordIndex = i + j * wordLength;
        // get the next word from the string
        string word = str.substr(nextWordIndex, wordLength);
        if (wordFrequencyMap.find(word) ==
            wordFrequencyMap.end()) {  // break if we don't need this word
          break;
        }

        wordsSeen[word]++;  // add the word to the 'wordsSeen' map

        // no need to process further if the word has higher frequency than required
        if (wordsSeen[word] > wordFrequencyMap[word]) {
          break;
        }

        if (j + 1 == wordsCount) {  // store index if we have found all the words
          resultIndices.push_back(i);
        }
      }
    }

    return resultIndices;
  }
};

int main(int argc, char *argv[]) {
  vector<int> result =
      WordConcatenation::findWordConcatenation("catfoxcat", vector<string>{"cat", "fox"});
  for (auto num : result) {
    cout << num << " ";
  }
  cout << endl;

  result = WordConcatenation::findWordConcatenation("catcatfoxfox", vector<string>{"cat", "fox"});
  for (auto num : result) {
    cout << num << " ";
  }
  cout << endl;
}
```



Time Complexity

The time complexity of the above algorithm will be $O(N * M * Len)$ where 'N' is the number of characters in the given string, 'M' is the total number of words, and 'Len' is the length of a word.

Space Complexity

The space complexity of the algorithm is $O(M)$ since at most, we will be storing all the words in the two **HashMaps**. In the worst case, we also need $O(N)$ space for the resulting list. So, the overall space complexity of the algorithm will be $O(M + N)$.