滑动窗口

笔记本:	0_leetcode		
创建时间:	2020/8/11 22:22	**更新时间:**	2020/8/25 11:06
作者:	415669592@qq.com		
URL:	https://www.educative.io/courses/grokking-the-coding-interview/7D5NNZ...		

滑动窗口

0. 介绍

关键词：在一个数组中**寻找或者计算连续的给定大小子数组。**

1. 入门

> 给定一个数组，计算连续K个元素的均值。
> Array: [1, 3, 2, 6, -1, 4, 1, 8, 2], K = 5
> Output: [2.2, 2.8, 2.4, 3.6, 2.8]

思路

> 不论做什么题，一开始别去找最好方法，直接想最简单的暴力方法。

1. 计算索引为0-4的元素均值
2. 计算1-5， 2-6.以此类推。

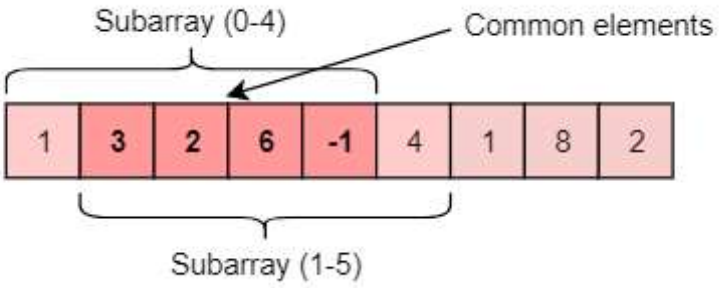
```
using namespace std;

#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;
    }
```

很明显，时间复杂度为O(N * K)

优化1-1——窗的视角

不难看出，我们在计算0-4的子数组后，计算1-5的子数组。我们发现1-4实际上我们已经计算过了。

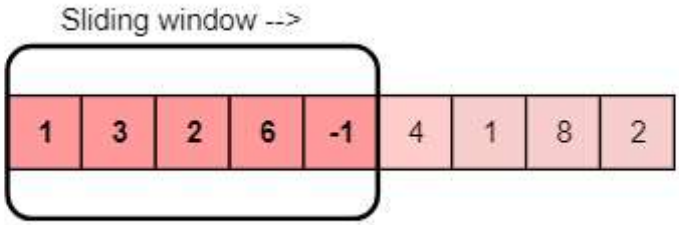


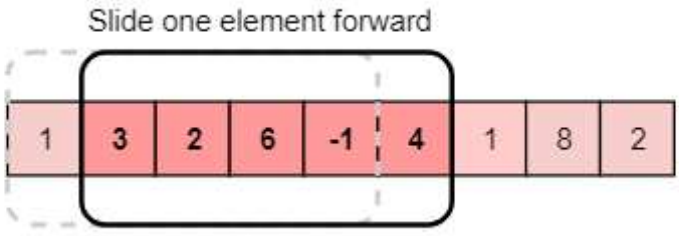
实际上，我们在计算1-5的时候，只需要减去0，再加上5就可以了。

`sum[1] = sum[0] - num[0] + num[5]`

写成通用代码就是

`sum[i] = sum[i-1] - num[i-1] + num[i+K-1]`






```
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;
}
```

复杂度已经被我们优化成了O(N)

优化1-2——sum的视角

我们站在更高的角度上来看待一下这个问题。不难看出这个问题实际上就是在针对 `array[i, j]` 做文章。

```
ResultType array(i, j){
    foreach(i, j, calc_method);
```

```
}
```

这道题里，实际上calc_method就是sum 。我们只要知道sum[i, j]，问题迎刃而解。

sum[i,j] = sum[0,j] - sum[0, i-1]

```
using namespace std;

#include <iostream>
#include <vector>

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

        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;
}
```

小结

关键字：1. 窗口 2. 累积

2. 实战

2.1 大于等于目标值的最短连续子数组长度

> 给定一个正数数组，给定一个目标值S。寻找大于目标值的最短连续子数组长度。

e.g.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].
```

e.g.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].
```

e.g.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].
```

思路

寻找差异性：和入门题相比，这里的size不是固定。是变动的。

1. O(N2)的暴力循环。因为在入门题中，已经谈到过过暴力解的重复计算问题。这里就不再展开了。
2. 站在窗口的角度上。

使用纸和笔模拟一遍。实际上，就是要寻找什么时候右边界该滑动，什么时候左边界该滑动。

[2] 不满足条件，右 边界滑动

[2 1] 不满足条件，右 边界滑动

[2 1 5] 满足条件，记录长度。左 边界滑动

[1 5] 不满足条件....

```
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;
}
```

3. 站在累积的角度上看，我们能否解决这个问题呢？可以发现，因为给定的size不固定，所以，我们需要去遍历所有的size，其本质最终和O(N2)的暴力遍历是一样的。

2-2 给定字符集的最长子串

> 给定字符集长度K，求最长子串。子串中只能出现K种字符，相同字符可以重复出现

e.g.1

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

e.g.2

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

e.g.3

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

思路

迅速寻找关键点，直接套用滑动窗口。

1. 子数组长度不固定。因此直接去寻找让滑动窗口滑动的方法。

以e.g.1为例

```
[a] 满足条件，右边界滑动，记录长度
[a r] 满足条件，右边界滑动，记录长度
[a r a] 同上
[a r a a] 满足条件，右边界滑动，记录长度
[a r a a c] 不满足条件，出现三种字符。左边界滑动
  [r a a c] 同上
    [a a c] 满足条件，右边界滑动，记录长度
    ...
```

问题变成了如何确定当前子数组的不同字符个数。相当于判断当前字符str[i]是否出现在我的窗口字符集中。
很明显，使用hashmap。

> 又因为字符集大小给定 (最多256个字符)，我们一般使用数组来做hashmap。
> （这个一般是优化点）

```
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;
}
```

2-3 摘水果

> 给定一个字符数组。每个字符代表一种水果。给定俩个篮子。每个篮子只能装一种水果。求俩个篮子可以最多装多少个水果。
> 可以从任意位置开始摘，但是不能跳过水果。知道不能摘为止（很明显地暗示了连续子数组的概念）

e.g.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']
```

e.g.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']
```

思路

直接套用滑动窗口的模式来做。

[A] 满足条件，右边界向右滑动

[A B] 满足条件，右边界向右滑动

[A B C] 不满足条件，左边界向左滑动

[B C] 满足条件，右边界向右滑动

[B C A] 不满足条件，右边界向右滑动

[C A] 满足条件，右边界向右滑动

[C A C] 满足条件，右边界向右滑动

不难发现，这道题和上题非常相似。仅仅是将K确定为了2而已。

```
using namespace std;

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

class MaxFruitCountOf2Types {
  public:
    static int findLength( const vector<char> &arr ) {
        int maxLength = 0;
        // TODO: Write your code here
        unordered_map<char, int> hashmap;
        int win_start = 0;
        for ( int win_end = 0; win_end < arr.size();  ) {
            hashmap[arr[win_end]]++;
            if ( hashmap.size() <= 2 ) {
                win_end++;
                int len = win_end - win_start;
                maxLength = std::max( maxLength, len );
            } else {
                hashmap[arr[win_start]]--;
                if ( hashmap[arr[win_start]] == 0 )
                    hashmap.erase( arr[win_start] );
                win_start++;
            }
        }
        return maxLength;
    }
};


int main() {
```

```
    vector<char> arr = { 'A','B','C','B','B','C' };
    int len = MaxFruitCountOf2Types::findLength( arr );
    cout << len << endl;
}
```

2-4 无重复字符的最长子串(leetcode-3)

> 无重复最长子串。

e.g.1

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

e.g.2

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

思路

算是leetcode上比较经典的一道滑动窗口题目。
直接套用滑动窗口模式来做。

[a] 满足条件，右边界滑动

[a a] 不满足条件，左边界滑动

[a b] 满足条件，右边界滑动

[a b c] 满足条件，右边界滑动

[a b c c] 不满足条件

[b c c]

[c c]

[c] 注释[1]

可以看到，这里的难点实际上变为了如何判断满足条件。即，如何统计只 统计当前窗口。（最好的办法就是在滑动的时候让我们的hashmap一起发生变动）

慢滑动版本。

```
using namespace std;

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

class NoRepeatSubstring {
  public:
    static int findLength( const string &str ) {
        int maxLength = 0;
        // TODO: Write your code here
        unordered_map<char, int> hashmap;
        int win_start = 0;
        for ( int win_end = 0; win_end < str.size(); win_end++ ) {
            hashmap[str[win_end]]++;
            while ( hashmap[str[win_end]] > 1 ) {
                hashmap[str[win_start]]--;
                win_start++;
            }
            maxLength = std::max( win_end + 1 - win_start, maxLength );
        }
        return maxLength;
    }
};
int main() {
    string str = "aabccbb";
    int len = NoRepeatSubstring::findLength( str );
    cout << len << endl;
    system( "pause" );
}
```

从注释[1]处，我们可以看到，当发生条件不满足时

[a b c c]
可以一步直接滑动到

[c]
即我们通过hashmap判断发生冲突时，左边界可以直接滑动到hashmap[curr_char] + 1的位置。这也正是leetcode中的做法。

2-5 替换给定字符，求最长子数组长度

> 给定字符串。最多可以替换不超过K个字符。寻找替换后的最长子数组长度

e.g.1

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

e.g.2

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

e.g.3

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

思路

还是套用滑动窗口的模式
假设K = 2

[a a b c c b b]

难点还是条件满足的判断。因此我们先分析条件。

1. 窗口存在一个主的重复字符。我们需要记录
2. 窗口长度 - 主重复字符长度 <= 2
3. 不难发现，我们主要关注的条件就是维护窗口的长度。即主字符串长度，很简单。频率就是窗口中主字符的长度。

[a] 满足条件，a的长度为1， 记录为主字符

[a a] 满足条件，a的长度为2， 记录为主字符

[a a b] 满足条件，a的长度为2， b的长度为1，记录a的长度。

[a a b c] 同上

[a a b c c] 条件2不满足，左窗口滑动。

[a b c c]

[a b c c b] 条件2不满足，左窗口滑动

[b c c b]

[b c c b b] b的字符串长度为3，更新为新的最长长度

> 解释[1]，注意滑动左边界。假设出现了
>
> [b c c b b d] 左边界滑动
>
> [c c b b d] 而此时maxRepeatLetterCount 依然是3.并不会缩小。因为没有出现更长的主串，窗口大小不会改变。所以不会影响结果值。大概意思就是窗口大小只会被扩张。而不会被缩小。

```
using namespace std;

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

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++;
            }// 注意这里的条件判断，我们并非严格的持有主字符频率。即在
shrink的时候，并不去修改maxRepeatLetterCount
            // 因为我们的目标是得到最大长度。所以只有出现新的更长的主字
符时，才会导致k + maxRepeatLetterCount增加，窗的长度才会发生变化。这里
```

```
对应解释[1]

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

        return maxLength;
    }
};

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

2-6 0替换1

> 给定01序列，给定K，可以最多替换K个0为1。求连续为1的子数组最大长度。

e.g.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 1s having length 6.
```

e.g.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 subarray of 1s having length 9.
```

思路

这道题和上道题非常相似。上道题维护的是一个主字符的最大长度。而这里的主字符就是1。实际上就是维护窗口中1的长度。

很简单，直接将上题中维护的hashmap变成一个int就可以了。

```cpp
using namespace std;

#include <iostream>
#include <vector>
#include <algorithm>

class ReplacingOnes {
  public:
    static int findLength( const vector<int> &arr, int k ) {
        int  maxLength = 0;
        // TODO: Write your code here
        int win_start = 0;
        int one_count = 0;
        for ( int win_end = 0; win_end < arr.size(); win_end++ ) {
            if ( arr[win_end] == 1 ) {
                one_count++;
            }
            int win_len = win_end + 1 - win_start;
            if ( win_len - one_count > k ) {
                if ( arr[win_start] == 1 )
                    one_count--;
                win_start++;
                win_len = win_end + 1 - win_start;
            }
            maxLength = std::max( maxLength, win_end + 1 - win_start );
        }
        return maxLength;
    }
};

int main() {
    vector<int> arr = { 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1 };
    int res = ReplacingOnes::findLength( arr, 2 );
    cout << res << endl;

    vector<int> arr2 = { 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1 };
    int res2 = ReplacingOnes::findLength( arr2, 3 );
    cout << res2 << endl;

}
```

3. 变型问题

3.1 next_permutation

> 给定一个主串和一个模式串，判断主串中的任意子串是否是模式串的permutation串。

e.g.1

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

思路

这题的思路如果没见过，还真想不到还是利用滑动窗口来做。

在主串中滑动，判断各个子串是否是模式串的permutation。难点还是判断is_permutation。

判断is_permutation实际上只要一个子串的所有字符集种类和个数都和pattern匹配，就是了。

滑动过程如下

[o]

[o i]

[o i d]

[o i d b]

[i d b]

[i d b c]

[d b c]

[d b c a]

[b c a]

[b c a f]

[c a f]

```
using namespace std;

#include <iostream>
```

```cpp
#include <string>
#include <unordered_map>

class StringPermutation {
  public:
    static bool findPermutation( const string &str, const string
&pattern ) {
        // TODO: Write your code here
        int win_start = 0;
        unordered_map<char, int> pattern_char_freq;
        for ( int i = 0; i < pattern.size(); i++ ) {
            pattern_char_freq[pattern[i]]++;
        }
        int matched = 0;
        for ( int win_end = 0; win_end < str.size(); win_end++ ) {
            char right_char = str[win_end];
            if ( pattern_char_freq.count( right_char ) != 0 ) {
                pattern_char_freq[right_char]--;
                if( pattern_char_freq[right_char] == 0 )
                    matched++;
            }
            if ( matched == pattern_char_freq.size() )
                return true;
            if ( win_end + 1 - win_start == pattern.size() ) {
                //shrink
                char left_char = str[win_start];
                if ( pattern_char_freq.count( left_char ) != 0 ) {
                    if ( pattern_char_freq[left_char] == 0 )
                        matched--;
                    pattern_char_freq[left_char]++;
                }
                win_start++;
            }
        }
        return false;
    }
};

int main() {
    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;
```

```
}
```

额外知识点

这道题的匹配规则值得学习。就是如何去匹配给定字符集种类，给定个数。使用hashmap去匹配。

> 大致思想如下，用每个字符去匹配，匹配成功，char_freq的value就减1。当value为0的时候，matched++。代表有一个种类被匹配成功了。

Ps：我自己的思路就是再自己做一个hashmap去比较匹配。但是复杂度是O(K)， K = hashmap.size()

3.2

> 给定一个主串一个模式串，在主串中找到所有模式串的异位词的索引

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

这里和上道题的思路非常像。

```
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
        unordered_map<char, int> char_freq;
        for ( int i = 0; i < pattern.size(); i++ ) {
            char_freq[pattern[i]]++;
        }
```

```
        int matched = 0;
        int win_start = 0;
        for ( int win_end = 0; win_end < str.size(); win_end++ ) {
            char right_char = str[win_end];
            if ( char_freq.find( right_char ) != char_freq.end() ) {
                char_freq[right_char]--;
                if ( char_freq[right_char] == 0 )
                    matched++;
            }

            if ( matched == char_freq.size() && win_end + 1 - win_start
== pattern.size() ) {
                resultIndices.push_back( win_start );
            }
            if ( win_end + 1 - win_start == pattern.size() ) {
                char left_char = str[win_start];
                if ( char_freq.find( left_char ) != char_freq.end() ) {
                    if ( char_freq[left_char] == 0 ) {
                        matched--;
                    }
                    char_freq[left_char]++;
                }
                win_start++;
            }
        }
        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 << " ";
    }
    cout << endl;

    system( "pause" );
```

```
}
```

值得学习的点

个人认为上俩道题非常值得学习。算是滑动窗口给定窗口大小K的范式解法。

1. 选取数据结构来维护窗口中的内容。这里选取的是hashmap。
2. 滑动窗口范式。俩种

```
int win_start = 0;
for(int win_end = 0; win_end < input.size(); win_end++)
{
    if(condition_is_true)
        expand win_end;
   [1] if(满足符合结果的条件 && win_end + 1 - win_start == K)
        result << calc();
    if(win_end + 1 - win_start == K)
        shrink win_start;
}
```

不难发现。除了开始外，在[1]处我们就维护了窗口大小为K的一个窗。每次expand win_end的时候，对我们维护的数据结构做记录。增加或者减少。在shrink win_start的时候，做相反的操作。
这样就能维护我们的窗在一个稳定的状态。

第二种

```
int win_start = -1;
for(int win_end = 0; win_end < input.size(); win_end++)
{
    if(condition_is_true)
        expand win_end;

    if(win_end + 1 - win_start > K)
        shrink win_start;
    [1] if(满足符合结果的条件 && win_end + 1 - win_start == K)
        result << calc();
}
```

但是这里要处理好多关于win_start = -1的起始位置问题的边界判断。比较麻烦。

简单地来说。对于第一种方法是expand到K的时候，去判断结果。
而第二种方法则是shrink到K的时候，再去判断。

4. 总结

简单来说，对于滑动窗口可以简单分为俩类。

1. 给定窗口的Size。这里我建议左边界shrink的条件是`win_end + 1 - win_start == Size` 。我自己喜欢用>但是，后来发现，使用==来处理的话，可以节省很多不必要的判断。
2. 不给定窗口的Size，这里一般问题都是去求复合条件的最大窗口大小。
3. 滑动窗口的模式算是比较固定的套路

```
int win_start = 0;
for(int win_end = 0; win_end < size; win_end++)
{
    if(condition_is_true)
        expand win_end;
    if(win_end + 1 - win_start == winSize)
        shrink win_start
}
```

4. 一般情况下，我们可能需要维护比如窗口的长度，比如窗口中某个字符的长度。窗口中字符的种类。以及窗口中字符的种类以及每个种类的长度。
   hashmap是你最好的选择。