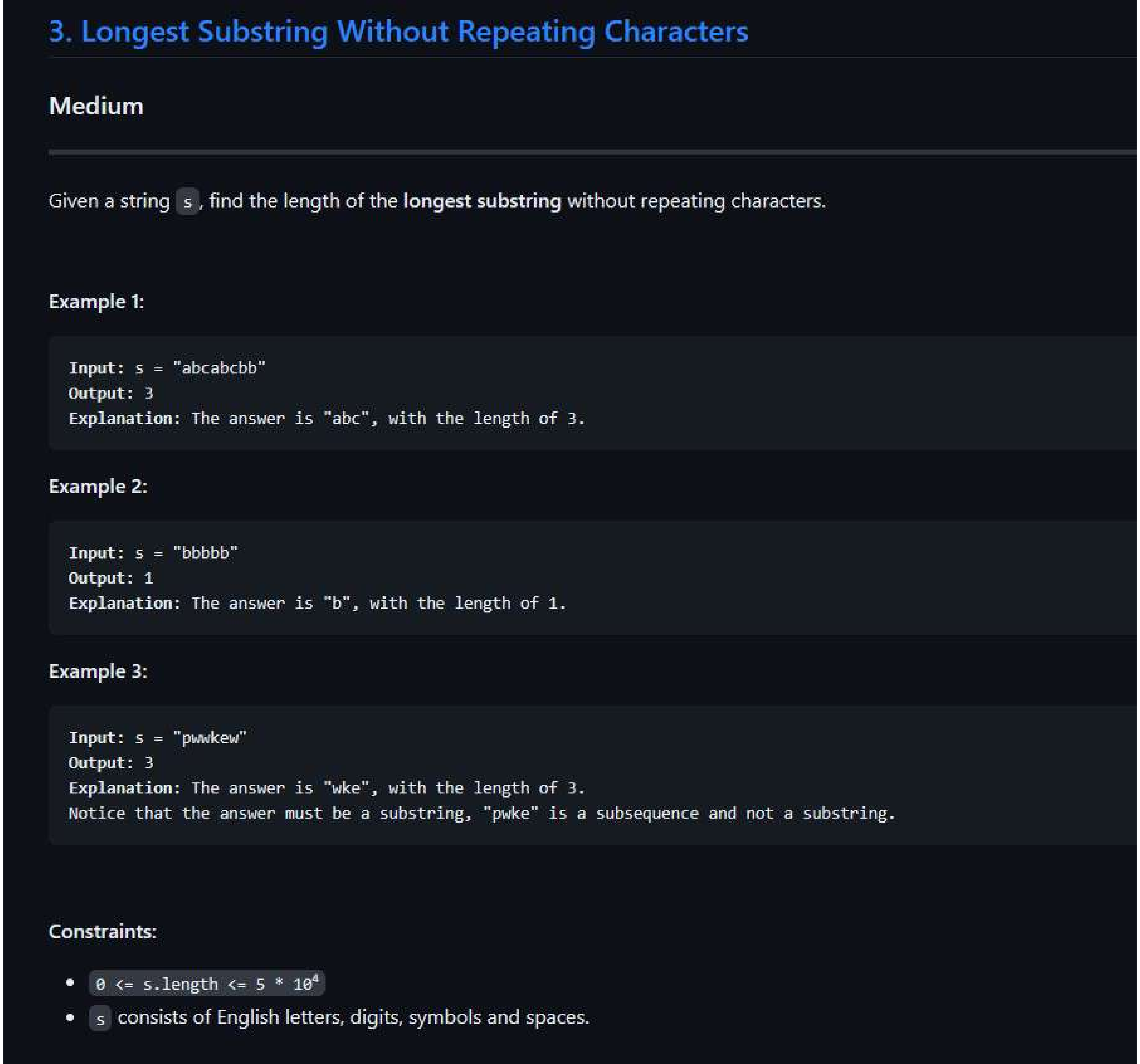


3. Longest Substring Without Repeating Characters

Medium

Given a string `s`, find the length of the **longest substring** without repeating characters.

Example 1:

```
Input: s = "abcabcbb"
Output: 3
Explanation: The answer is "abc", with the length of 3.
```

Example 2:

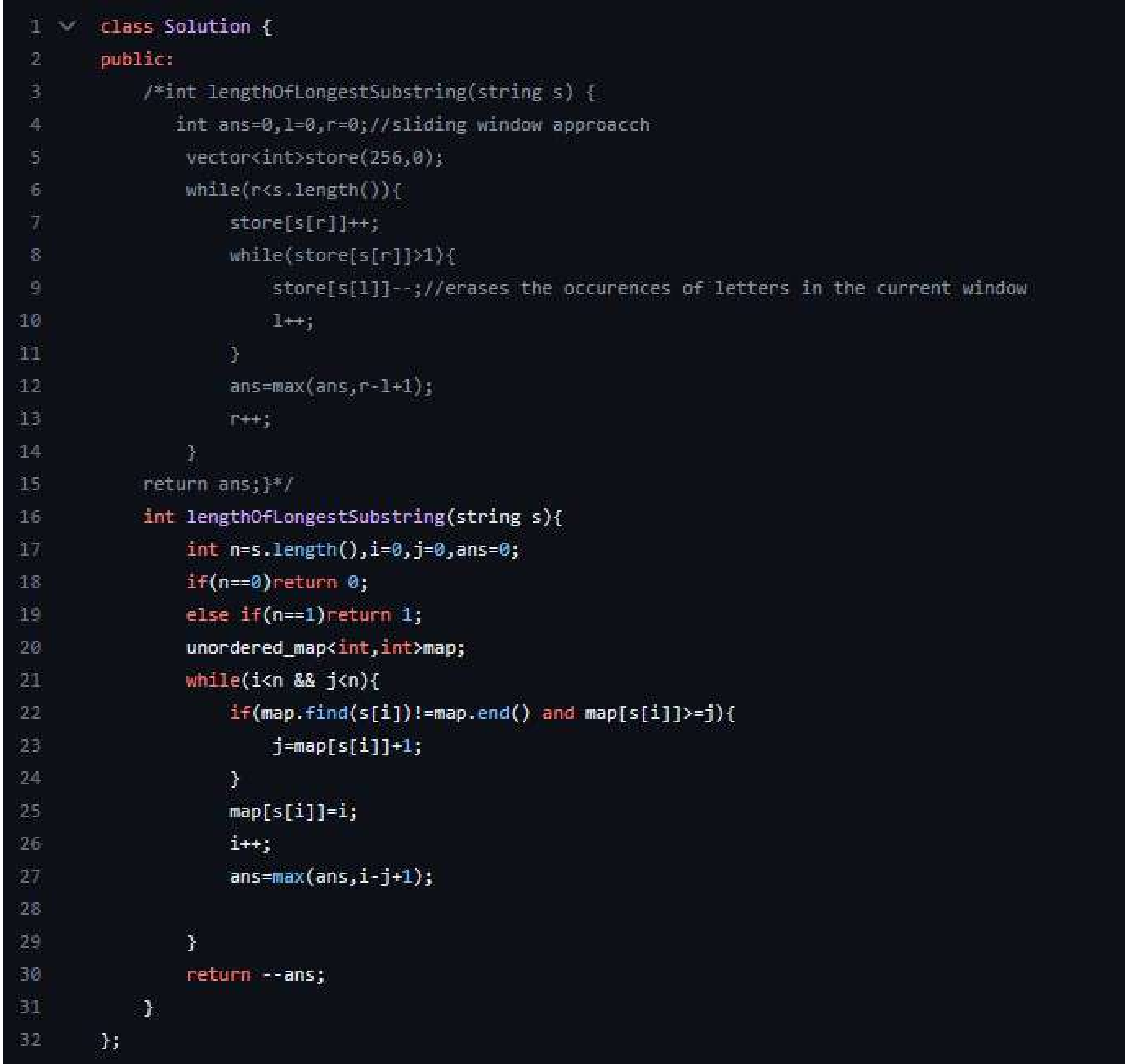
```
Input: s = "bbbbb"
Output: 1
Explanation: The answer is "b", with the length of 1.
```

Example 3:

```
Input: s = "pwwkew"
Output: 3
Explanation: The answer is "wke", with the length of 3.
Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.
```

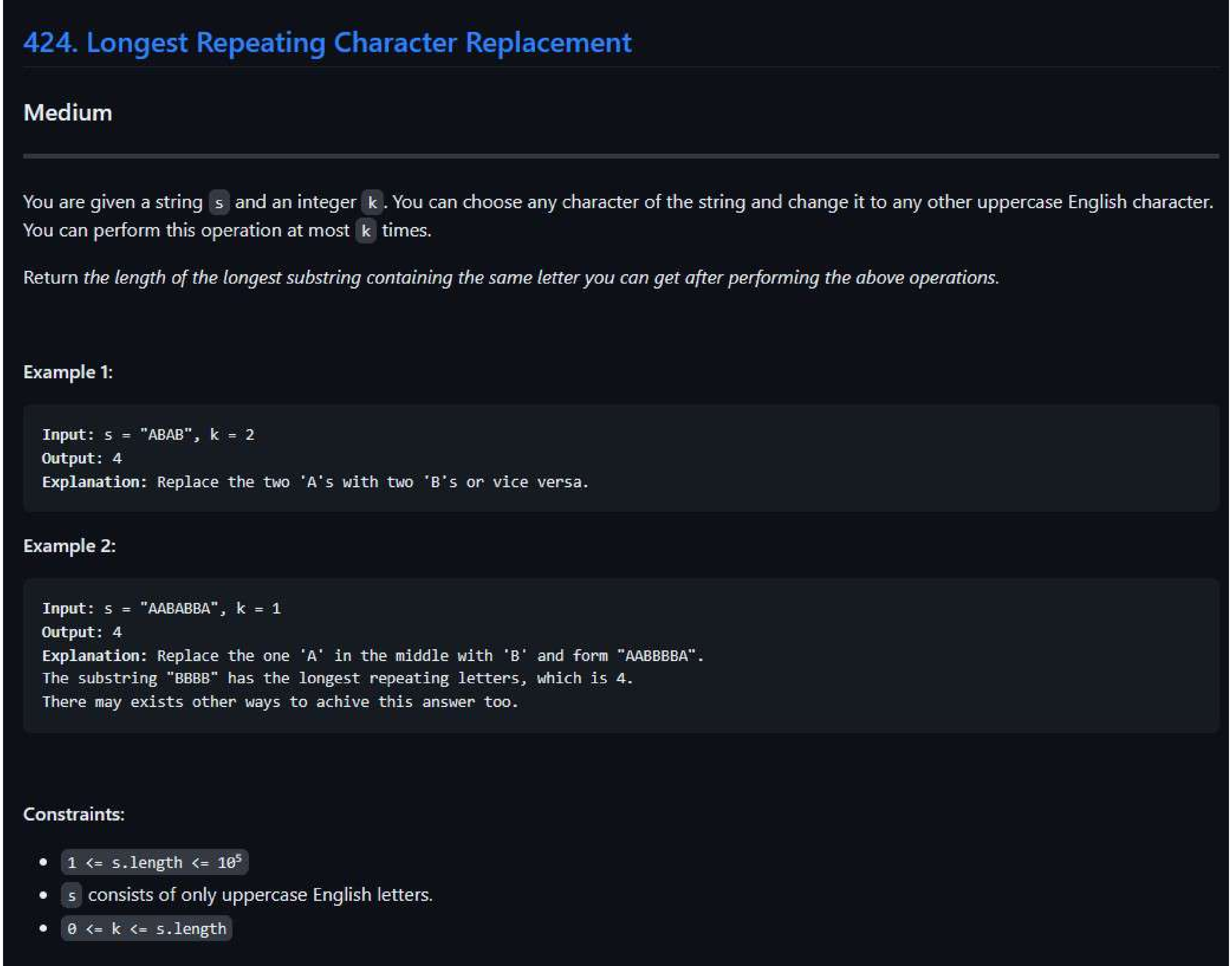
Constraints:

- $0 <= s.length <= 5 * 10^4$
- `s` consists of English letters, digits, symbols and spaces.



```cpp
class Solution {
public:
    /*int lengthOfLongestSubstring(string s) {
       int ans=0,l=0,r=0;//sliding window approacch
        vector<int>store(256,0);
        while(r<s.length()){
            store[s[r]]++;
            while(store[s[r]]>1){
                store[s[l]]--;//erases the occurences of letters in the current window
                l++;
            }
            ans=max(ans,r-l+1);
            r++;
        }
    return ans;}*/
    int lengthOfLongestSubstring(string s){
        int n=s.length(),i=0,j=0,ans=0;
        if(n==0)return 0;
        else if(n==1)return 1;
        unordered_map<int,int>map;
        while(i<n && j<n){
            if(map.find(s[i])!=map.end() and map[s[i]]>=j){
                j=map[s[i]]+1;
            }
            map[s[i]]=i;
            i++;
            ans=max(ans,i-j+1);

        }
        return --ans;
    }
};
```



424. Longest Repeating Character Replacement

Medium

You are given a string `s` and an integer `k`. You can choose any character of the string and change it to any other uppercase English character. You can perform this operation at most `k` times.

Return *the length of the longest substring containing the same letter you can get after performing the above operations.*

Example 1:

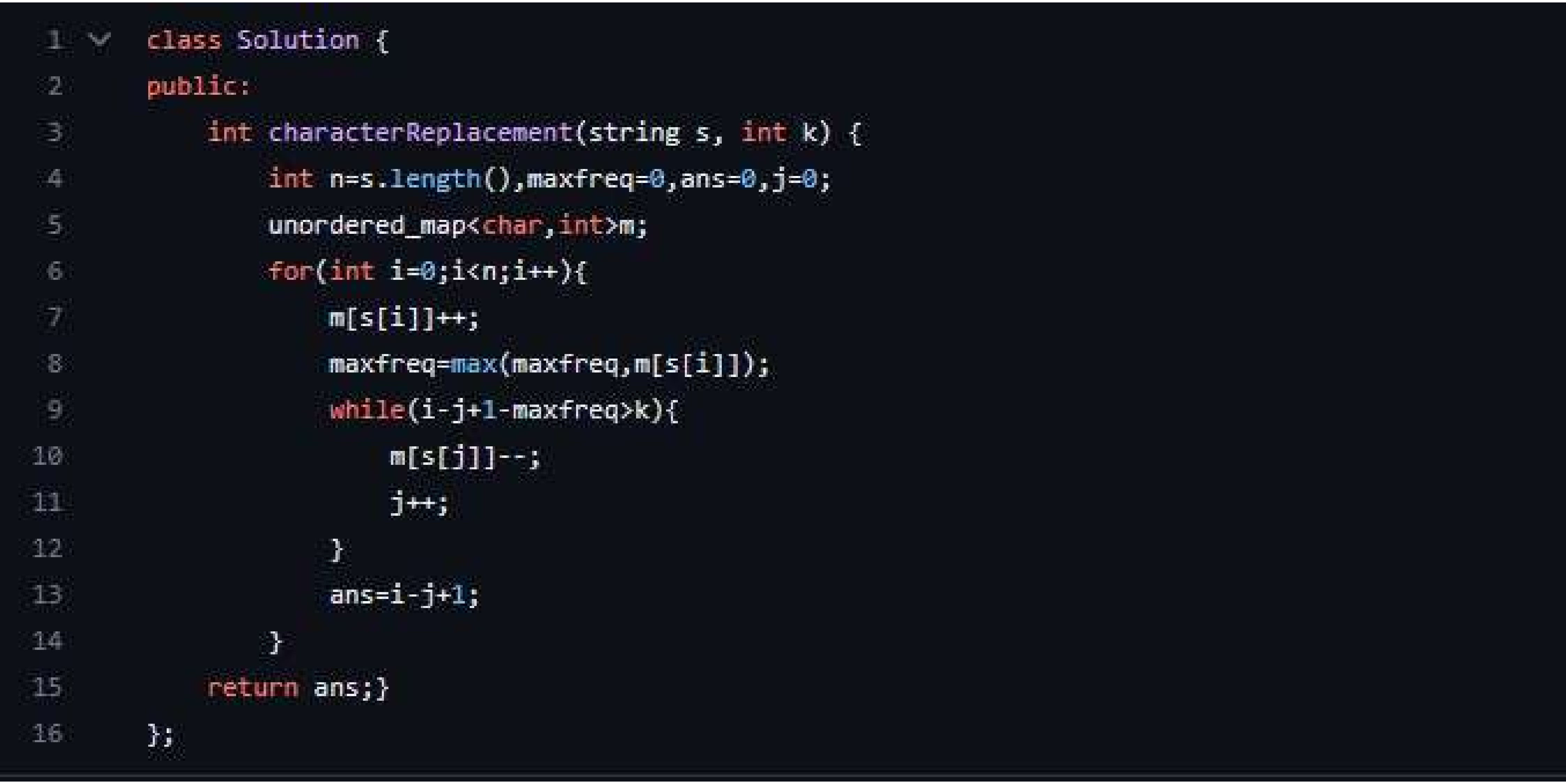
```
Input: s = "ABAB", k = 2
Output: 4
Explanation: Replace the two 'A's with two 'B's or vice versa.
```

Example 2:

```
Input: s = "AABABBA", k = 1
Output: 4
Explanation: Replace the one 'A' in the middle with 'B' and form "AABBBBA".
The substring "BBBB" has the longest repeating letters, which is 4.
There may exists other ways to achive this answer too.
```

Constraints:

- $1 <= s.length <= 10^5$
- `s` consists of only uppercase English letters.
- `0 <= k <= s.length`



```cpp
class Solution {
public:
    int characterReplacement(string s, int k) {
        int n=s.length(),maxfreq=0,ans=0,j=0;
        unordered_map<char,int>m;
        for(int i=0;i<n;i++){
            m[s[i]]++;
            maxfreq=max(maxfreq,m[s[i]]);
            while(i-j+1-maxfreq>k){
                m[s[j]]--;
                j++;
            }
            ans=i-j+1;
        }
    return ans;}
};
```

713. Subarray Product Less Than K

Medium

Given an array of integers `nums` and an integer `k`, return *the number of contiguous subarrays where the product of all the elements in the subarray is strictly less than* `k`.

Example 1:

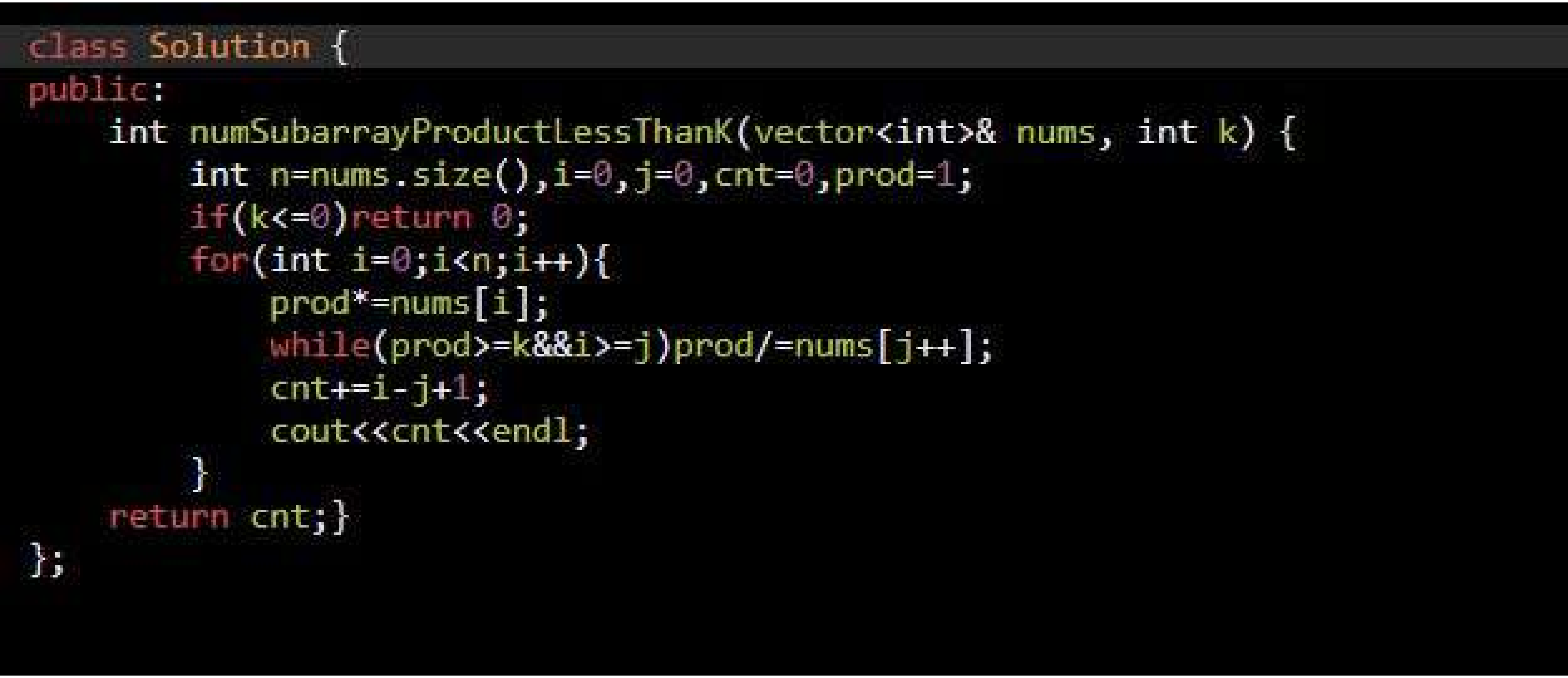
```
Input: nums = [10,5,2,6], k = 100
Output: 8
Explanation: The 8 subarrays that have product less than 100 are:
[10], [5], [2], [6], [10, 5], [5, 2], [2, 6], [5, 2, 6]
Note that [10, 5, 2] is not included as the product of 100 is not strictly less than k.
```

Example 2:

```
Input: nums = [1,2,3], k = 0
Output: 0
```

Constraints:

- $1 <= nums.length <= 3 * 10^4$
- `1 <= nums[i] <= 1000`
- $0 <= k <= 10^6$



```
class Solution {
public:
    int numSubarrayProductLessThanK(vector<int>& nums, int k) {
        int n=nums.size(),i=0,j=0,cnt=0,prod=1;
        if(k<=0)return 0;
        for(int i=0;i<n;i++){
            prod*=nums[i];
            while(prod>=k&&i>=j)prod/=nums[j++];
            cnt+=i-j+1;
            cout<<cnt<<endl;
        }
    return cnt;}
};
```

1004. Max Consecutive Ones III

Medium

Given a binary array `nums` and an integer `k`, return *the maximum number of consecutive* `1`*'s in the array if you can flip at most* `k` `0`'s.

Example 1:

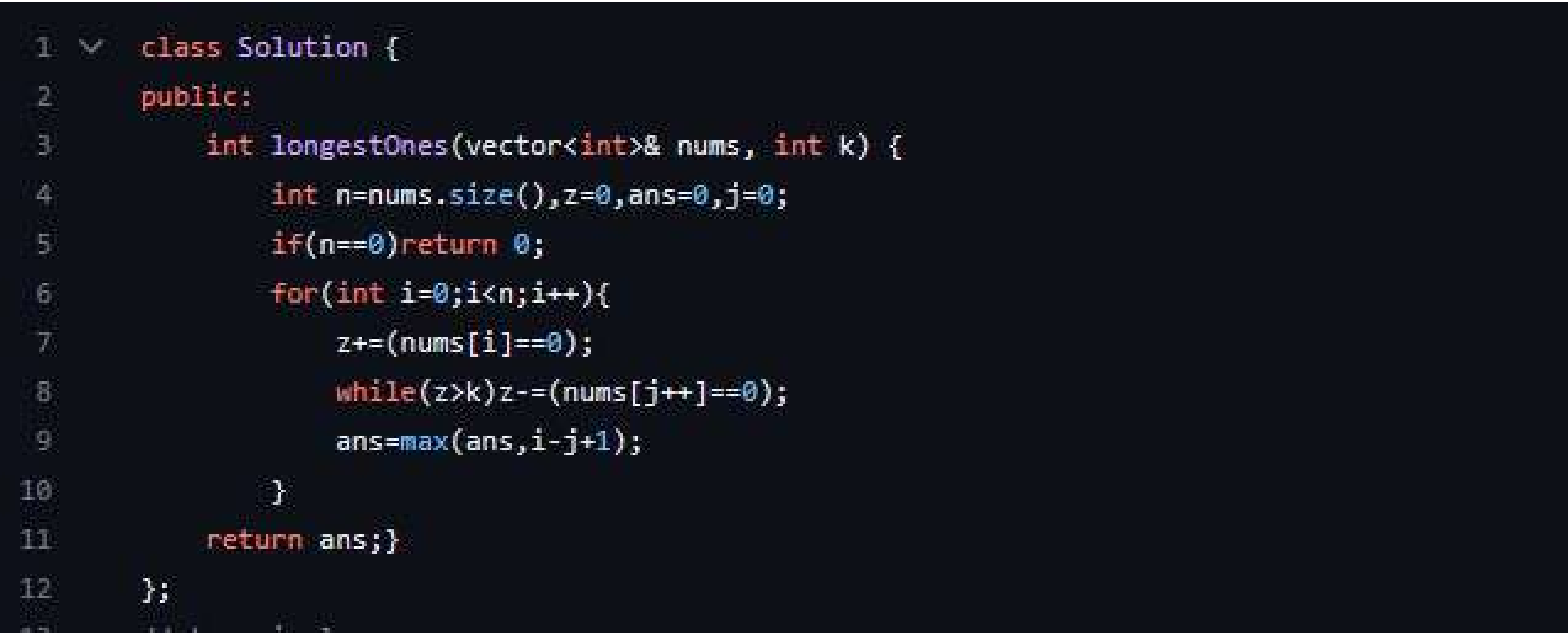
```
Input: nums = [1,1,1,0,0,0,1,1,1,1,0], k = 2
Output: 6
Explanation: [1,1,1,0,0,1,1,1,1,1,1]
Bolded numbers were flipped from 0 to 1. The longest subarray is underlined.
```

Example 2:

```
Input: nums = [0,0,1,1,0,0,1,1,1,0,1,1,0,0,0,1,1,1,1], k = 3
Output: 10
Explanation: [0,0,1,1,1,1,1,1,1,1,1,1,0,0,0,1,1,1,1]
Bolded numbers were flipped from 0 to 1. The longest subarray is underlined.
```

Constraints:

- `1 <= nums.length <= 10`^{5}
- `nums[i]` is either `0` or `1`.
- `0 <= k <= nums.length`



```
class Solution {
public:
    int longestOnes(vector<int>& nums, int k) {
        int n=nums.size(),z=0,ans=0,j=0;
        if(n==0)return 0;
        for(int i=0;i<n;i++){
            z+=(nums[i]==0);
            while(z>k)z-=(nums[j++]==0);
            ans=max(ans,i-j+1);
        }
    return ans;}
};
```

1208. Get Equal Substrings Within Budget

Medium

You are given two strings `s` and `t` of the same length and an integer `maxCost`.

You want to change `s` to `t`. Changing the `i`th character of `s` to `i`th character of `t` costs `|s[i] - t[i]|` (i.e., the absolute difference between the ASCII values of the characters).

Return *the maximum length of a substring of* `s` *that can be changed to be the same as the corresponding substring of* `t` *with a cost less than or equal to* `maxCost`. If there is no substring from `s` that can be changed to its corresponding substring from `t`, return `0`.

Example 1:

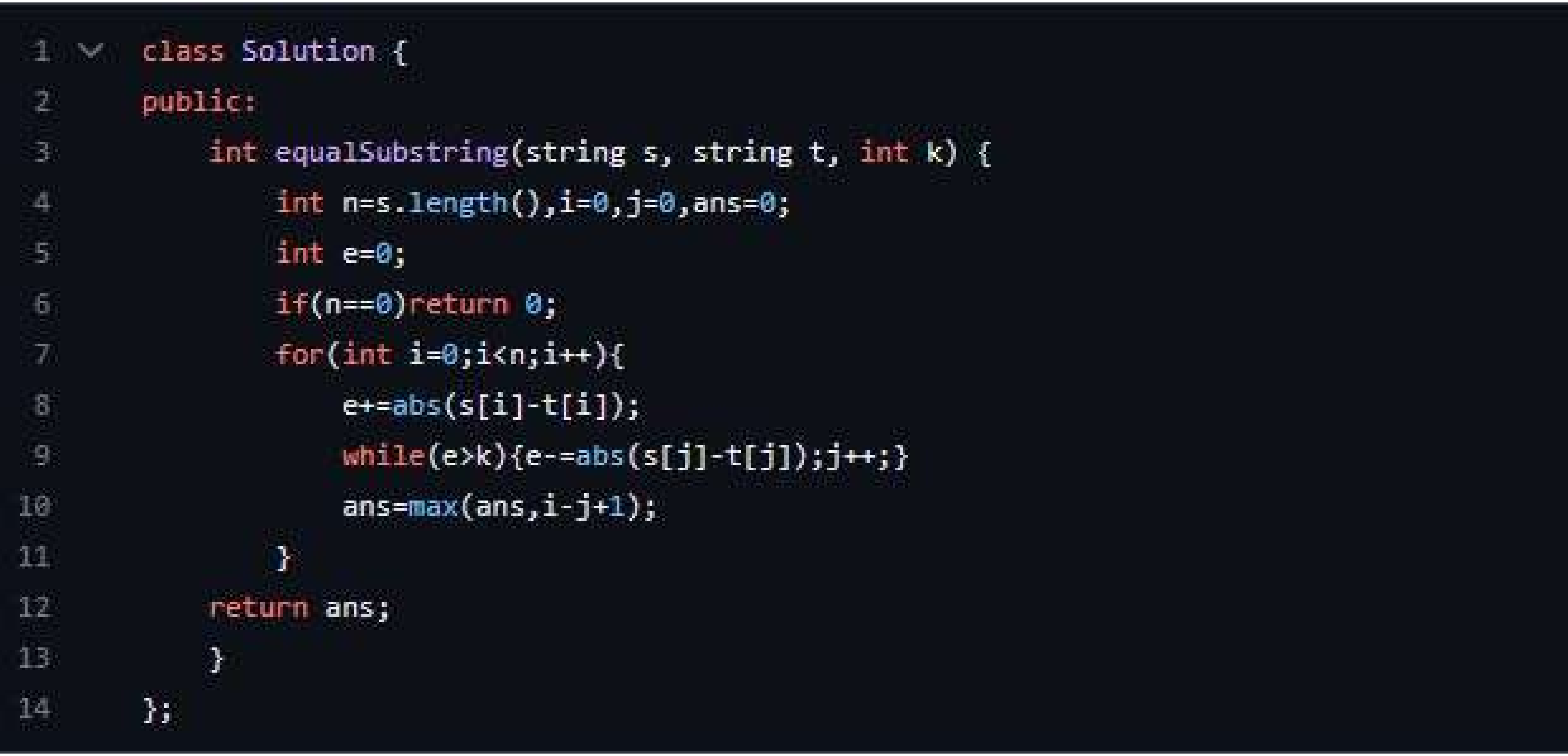
```
Input: s = "abcd", t = "bcdf", maxCost = 3
Output: 3
Explanation: "abc" of s can change to "bcd".
That costs 3, so the maximum length is 3.
```

Example 2:

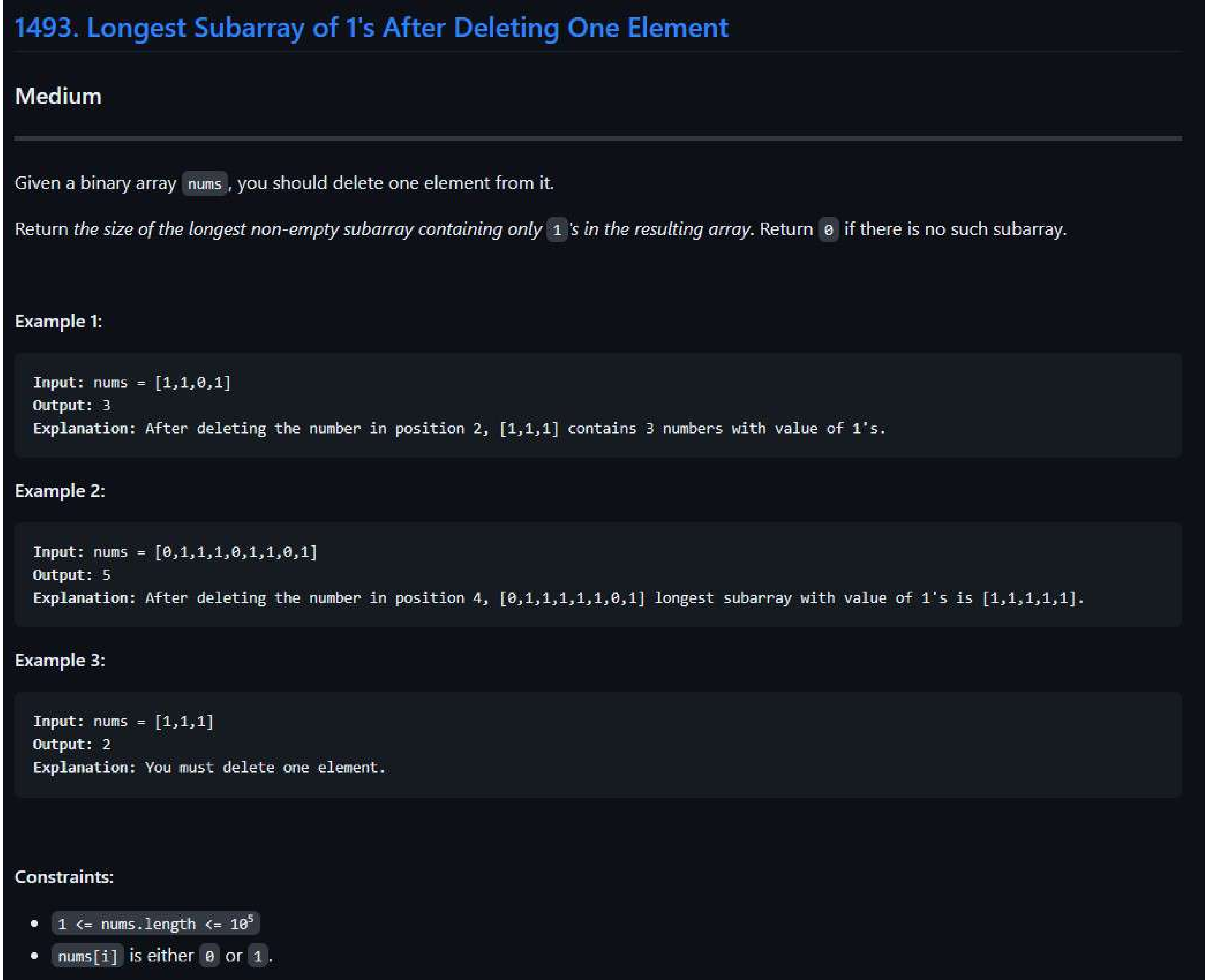
```
Input: s = "abcd", t = "cdef", maxCost = 3
Output: 1
Explanation: Each character in s costs 2 to change to character in t,  so the maximum length is 1.
```

Example 3:

```
Input: s = "abcd", t = "acde", maxCost = 0
Output: 1
Explanation: You cannot make any change, so the maximum length is 1.
```



```
class Solution {
public:
    int equalSubstring(string s, string t, int k) {
        int n=s.length(),i=0,j=0,ans=0;
        int e=0;
        if(n==0)return 0;
        for(int i=0;i<n;i++){
            e+=abs(s[i]-t[i]);
            while(e>k){e-=abs(s[j]-t[j]);j++;}
            ans=max(ans,i-j+1);
        }
    return ans;
    }
};
```



1493. Longest Subarray of 1's After Deleting One Element

Medium

Given a binary array `nums`, you should delete one element from it.

Return *the size of the longest non-empty subarray containing only* `1`*'s in the resulting array*. Return `0` if there is no such subarray.

Example 1:

```
Input: nums = [1,1,0,1]
Output: 3
Explanation: After deleting the number in position 2, [1,1,1] contains 3 numbers with value of 1's.
```

Example 2:

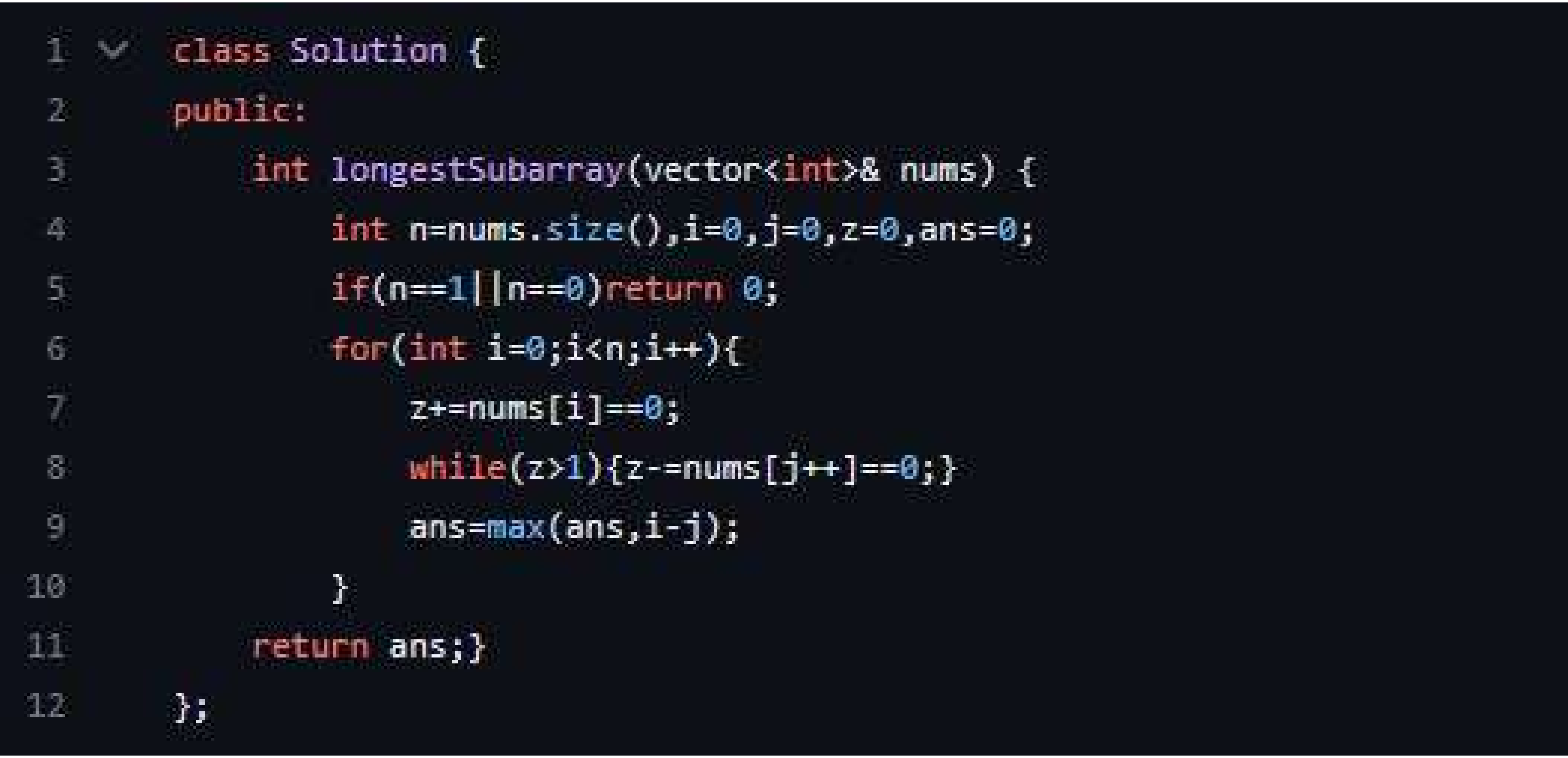
```
Input: nums = [0,1,1,1,0,1,1,0,1]
Output: 5
Explanation: After deleting the number in position 4, [0,1,1,1,1,1,0,1] longest subarray with value of 1's is [1,1,1,1,1].
```

Example 3:

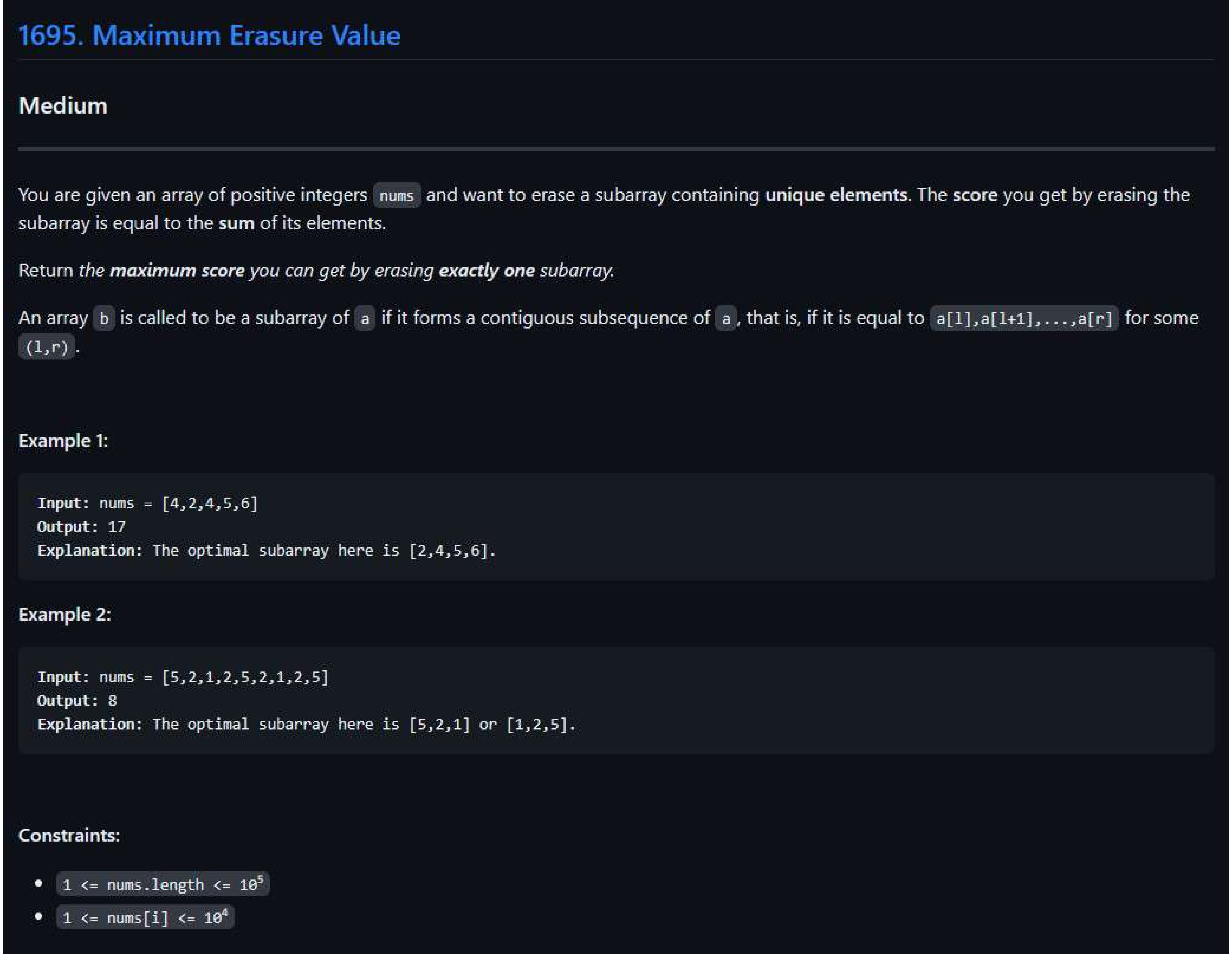
```
Input: nums = [1,1,1]
Output: 2
Explanation: You must delete one element.
```

Constraints:

- $1 <= nums.length <= 10^5$
- `nums[i]` is either `0` or `1`.



```cpp
class Solution {
public:
    int longestSubarray(vector<int>& nums) {
        int n=nums.size(),i=0,j=0,z=0,ans=0;
        if(n==1||n==0)return 0;
        for(int i=0;i<n;i++){
            z+=nums[i]==0;
            while(z>1){z-=nums[j++]==0;}
            ans=max(ans,i-j);
        }
    return ans;}
};
```



1695. Maximum Erasure Value

Medium

You are given an array of positive integers `nums` and want to erase a subarray containing **unique elements**. The **score** you get by erasing the subarray is equal to the **sum** of its elements.

Return *the **maximum score** you can get by erasing **exactly one** subarray.*

An array `b` is called to be a subarray of `a` if it forms a contiguous subsequence of `a`, that is, if it is equal to `a[l],a[l+1],...,a[r]` for some `(l,r)`.

Example 1:

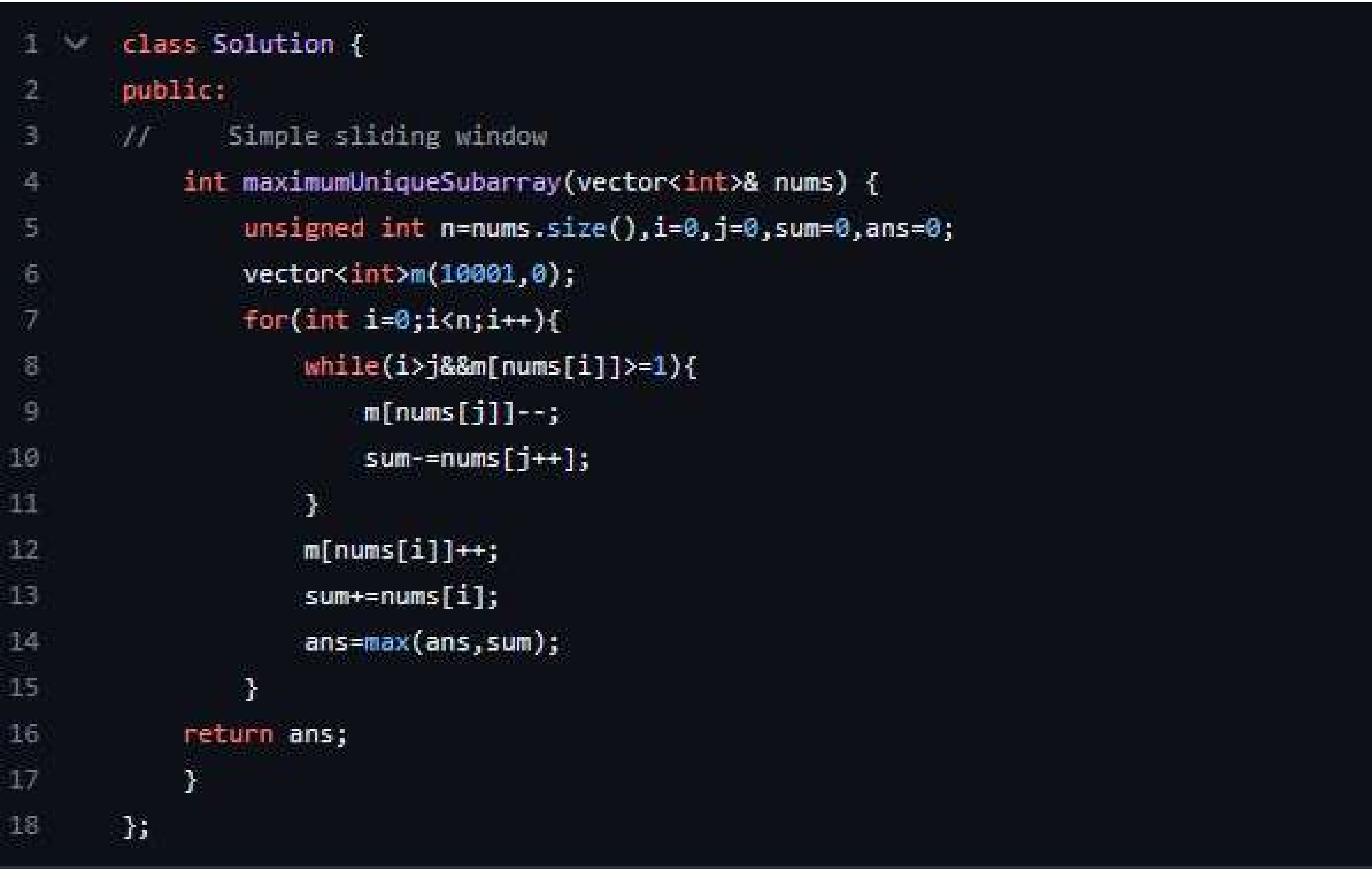
```
Input: nums = [4,2,4,5,6]
Output: 17
Explanation: The optimal subarray here is [2,4,5,6].
```

Example 2:

```
Input: nums = [5,2,1,2,5,2,1,2,5]
Output: 8
Explanation: The optimal subarray here is [5,2,1] or [1,2,5].
```

Constraints:

- $1 <= nums.length <= 10^5$
- $1 <= nums[i] <= 10^4$



```
class Solution {
public:
//      Simple sliding window
    int maximumUniqueSubarray(vector<int>& nums) {
        unsigned int n=nums.size(),i=0,j=0,sum=0,ans=0;
        vector<int>m(10001,0);
        for(int i=0;i<n;i++){
            while(i>j&&m[nums[i]]>=1){
                m[nums[j]]--;
                sum-=nums[j++];
            }
            m[nums[i]]++;
            sum+=nums[i];
            ans=max(ans,sum);
        }
    return ans;
    }
};
```

1838. Frequency of the Most Frequent Element

Medium

The **frequency** of an element is the number of times it occurs in an array.

You are given an integer array `nums` and an integer `k`. In one operation, you can choose an index of `nums` and increment the element at that index by `1`.

Return *the **maximum possible frequency** of an element after performing **at most** `k` operations.*

Example 1:

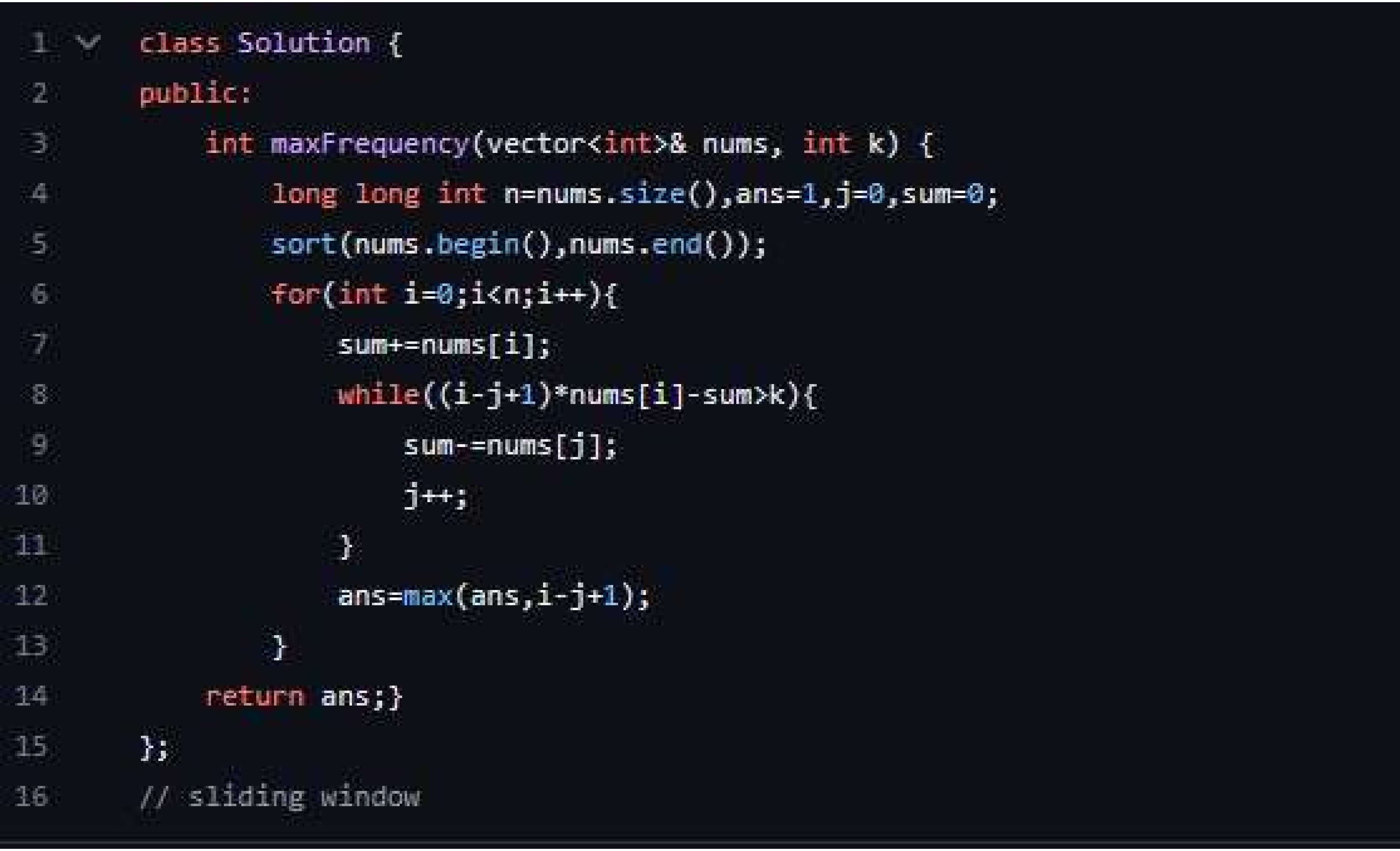
```
Input: nums = [1,2,4], k = 5
Output: 3
Explanation: Increment the first element three times and the second element two times to make nums = [4,4,4].
4 has a frequency of 3.
```

Example 2:

```
Input: nums = [1,4,8,13], k = 5
Output: 2
Explanation: There are multiple optimal solutions:
- Increment the first element three times to make nums = [4,4,8,13]. 4 has a frequency of 2.
- Increment the second element four times to make nums = [1,8,8,13]. 8 has a frequency of 2.
- Increment the third element five times to make nums = [1,4,13,13]. 13 has a frequency of 2.
```

Example 3:

```
Input: nums = [3,9,6], k = 2
Output: 1
```



```cpp
class Solution {
public:
    int maxFrequency(vector<int>& nums, int k) {
        long long int n=nums.size(),ans=1,j=0,sum=0;
        sort(nums.begin(),nums.end());
        for(int i=0;i<n;i++){
            sum+=nums[i];
            while((i-j+1)*nums[i]-sum>k){
                sum-=nums[j];
                j++;
            }
            ans=max(ans,i-j+1);
        }
    return ans;}
};
// sliding window
```

2009. Minimum Number of Operations to Make Array Continuous

Hard

You are given an integer array `nums`. In one operation, you can replace **any** element in `nums` with **any** integer.

`nums` is considered **continuous** if both of the following conditions are fulfilled:

- All elements in `nums` are **unique**.
- The difference between the **maximum** element and the **minimum** element in `nums` equals `nums.length - 1`.

For example, `nums = [4, 2, 5, 3]` is **continuous**, but `nums = [1, 2, 3, 5, 6]` is **not continuous**.

Return *the **minimum** number of operations to make* `nums` ***continuous***.

Example 1:

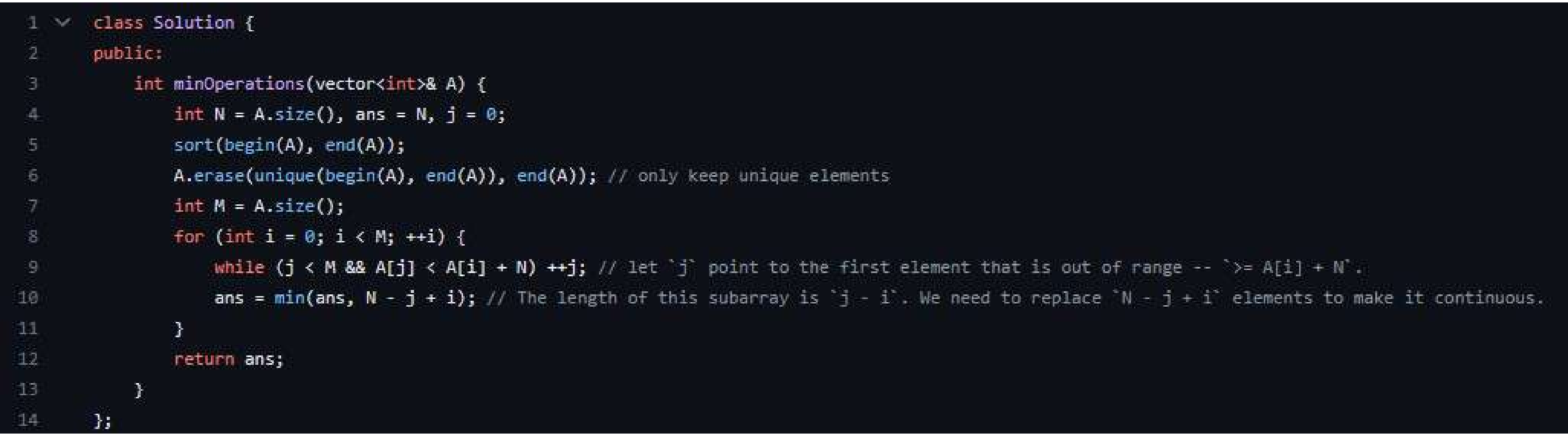
```
Input: nums = [4,2,5,3]
Output: 0
Explanation: nums is already continuous.
```

Example 2:

```
Input: nums = [1,2,3,5,6]
Output: 1
Explanation: One possible solution is to change the last element to 4.
The resulting array is [1,2,3,5,4], which is continuous.
```

Example 3:

```
Input: nums = [1,10,100,1000]
Output: 3
Explanation: One possible solution is to:
- Change the second element to 2.
```



```cpp
class Solution {
public:
    int minOperations(vector<int>& A) {
        int N = A.size(), ans = N, j = 0;
        sort(begin(A), end(A));
        A.erase(unique(begin(A), end(A)), end(A)); // only keep unique elements
        int M = A.size();
        for (int i = 0; i < M; ++i) {
            while (j < M && A[j] < A[i] + N) ++j; // let `j` point to the first element that is out of range -- `>= A[i] + N`.
            ans = min(ans, N - j + i); // The length of this subarray is `j - i`. We need to replace `N - j + i` elements to make it continuous.
        }
        return ans;
    }
};
```

2024. Maximize the Confusion of an Exam

Medium

A teacher is writing a test with `n` true/false questions, with `'T'` denoting true and `'F'` denoting false. He wants to confuse the students by **maximizing** the number of **consecutive** questions with the **same** answer (multiple trues or multiple falses in a row).

You are given a string `answerKey`, where `answerKey[i]` is the original answer to the `i`th question. In addition, you are given an integer `k`, the maximum number of times you may perform the following operation:

- Change the answer key for any question to `'T'` or `'F'` (i.e., set `answerKey[i]` to `'T'` or `'F'`).

Return *the **maximum** number of consecutive* `'T'`s or `'F'`*s in the answer key after performing the operation at most* `k` *times.*

Example 1:

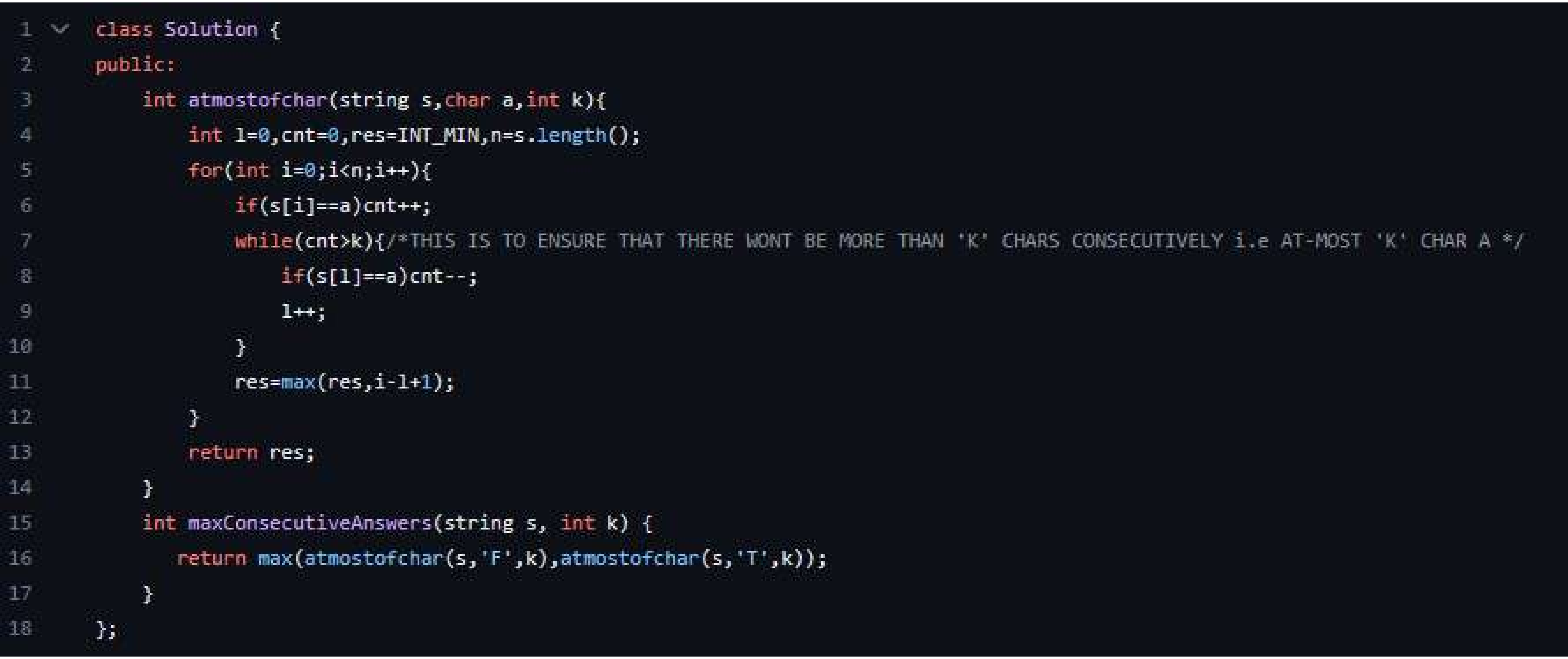
```
Input: answerKey = "TTFF", k = 2
Output: 4
Explanation: We can replace both the 'F's with 'T's to make answerKey = "TTTT".
There are four consecutive 'T's.
```

Example 2:

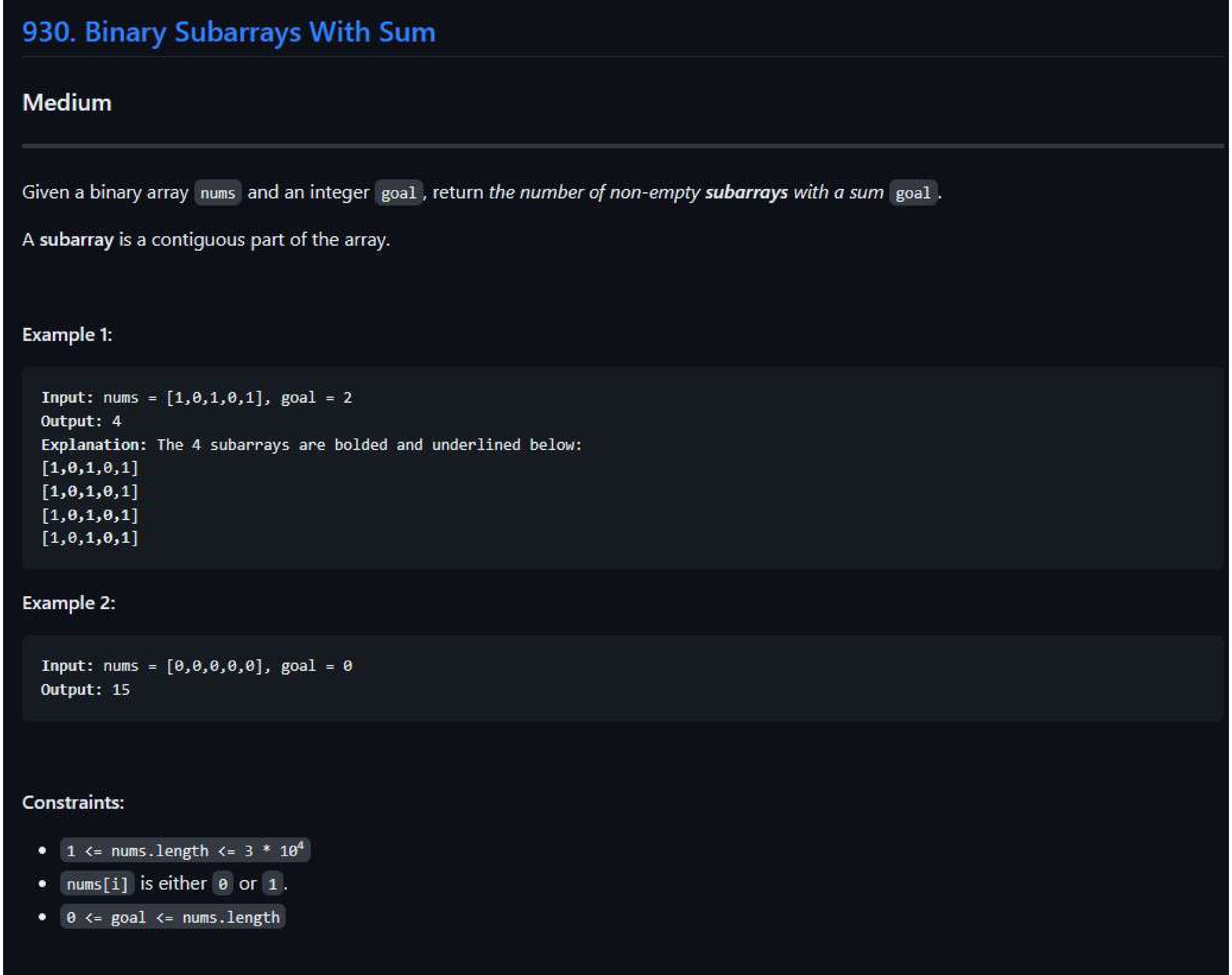
```
Input: answerKey = "TFFT", k = 1
Output: 3
Explanation: We can replace the first 'T' with an 'F' to make answerKey = "FFFT".
Alternatively, we can replace the second 'T' with an 'F' to make answerKey = "TFFF".
In both cases, there are three consecutive 'F's.
```

Example 3:

```
Input: answerKey = "TTFTTFTT", k = 1
Output: 5
Explanation: We can replace the first 'F' to make answerKey = "TTTTTFTT"
```



```cpp
class Solution {
public:
    int atmostofchar(string s,char a,int k){
        int l=0,cnt=0,res=INT_MIN,n=s.length();
        for(int i=0;i<n;i++){
            if(s[i]==a)cnt++;
            while(cnt>k){/*THIS IS TO ENSURE THAT THERE WONT BE MORE THAN 'K' CHARS CONSECUTIVELY i.e AT-MOST 'K' CHAR A */
                if(s[l]==a)cnt--;
                l++;
            }
            res=max(res,i-l+1);
        }
        return res;
    }
    int maxConsecutiveAnswers(string s, int k) {
       return max(atmostofchar(s,'F',k),atmostofchar(s,'T',k));
    }
};
```



930. Binary Subarrays With Sum

Medium

Given a binary array `nums` and an integer `goal`, return *the number of non-empty* ***subarrays*** *with a sum* `goal`.

A **subarray** is a contiguous part of the array.

Example 1:

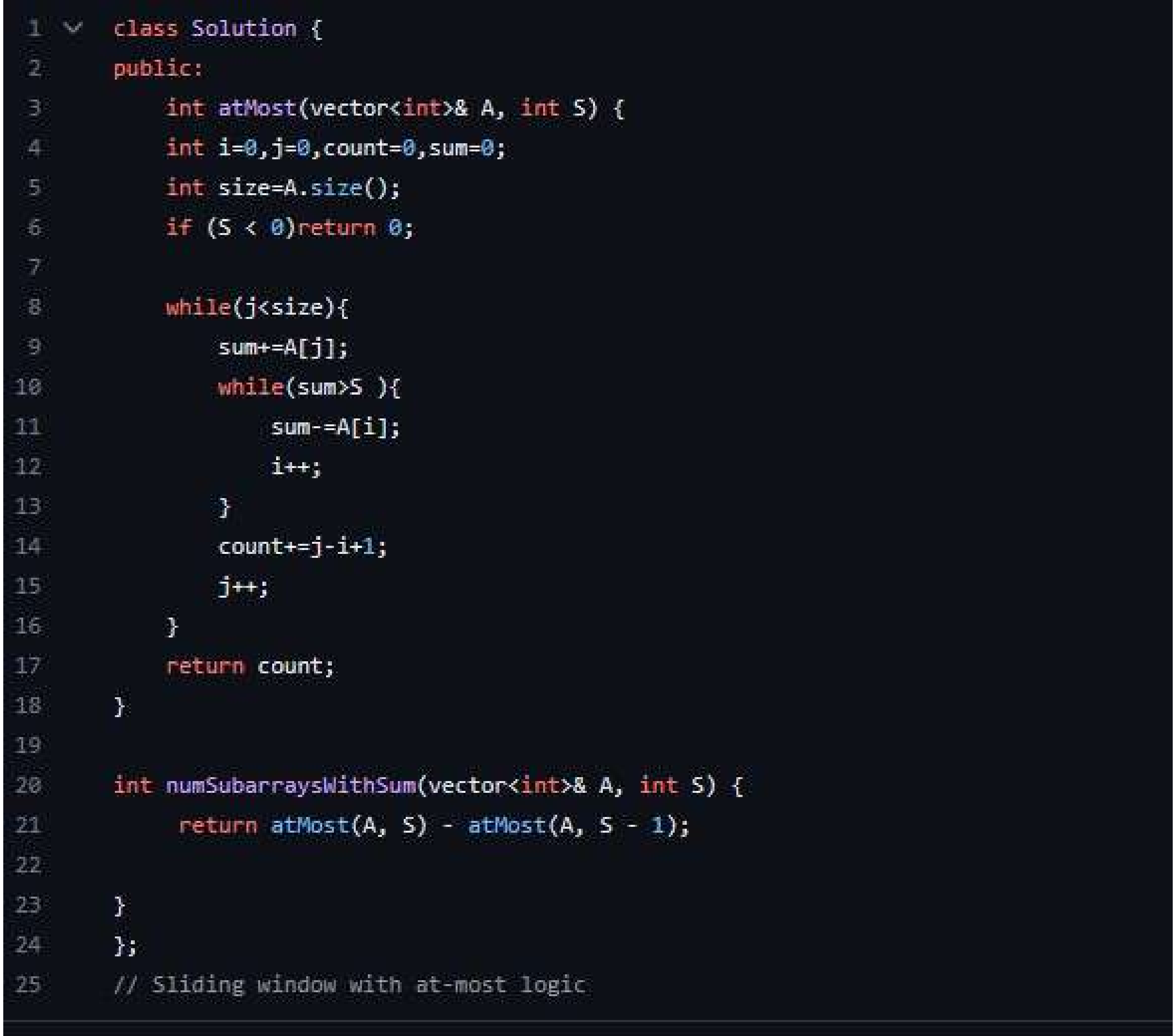
```
Input: nums = [1,0,1,0,1], goal = 2
Output: 4
Explanation: The 4 subarrays are bolded and underlined below:
[1,0,1,0,1]
[1,0,1,0,1]
[1,0,1,0,1]
[1,0,1,0,1]
```

Example 2:

```
Input: nums = [0,0,0,0,0], goal = 0
Output: 15
```

Constraints:

- `1 <= nums.length <= 3 * 10`^{4}
- `nums[i]` is either `0` or `1`.
- `0 <= goal <= nums.length`



```cpp
class Solution {
public:
    int atMost(vector<int>& A, int S) {
    int i=0,j=0,count=0,sum=0;
    int size=A.size();
    if (S < 0)return 0;

    while(j<size){
        sum+=A[j];
        while(sum>S ){
            sum-=A[i];
            i++;
        }
        count+=j-i+1;
        j++;
    }
    return count;
}

int numSubarraysWithSum(vector<int>& A, int S) {
     return atMost(A, S) - atMost(A, S - 1);

}
};
// Sliding window with at-most logic
```

992. Subarrays with K Different Integers

Hard

Given an integer array `nums` and an integer `k`, return *the number of **good subarrays** of* `nums`.

A **good array** is an array where the number of different integers in that array is exactly `k`.

- For example, `[1,2,3,1,2]` has `3` different integers: `1`, `2`, and `3`.

A **subarray** is a **contiguous** part of an array.

Example 1:

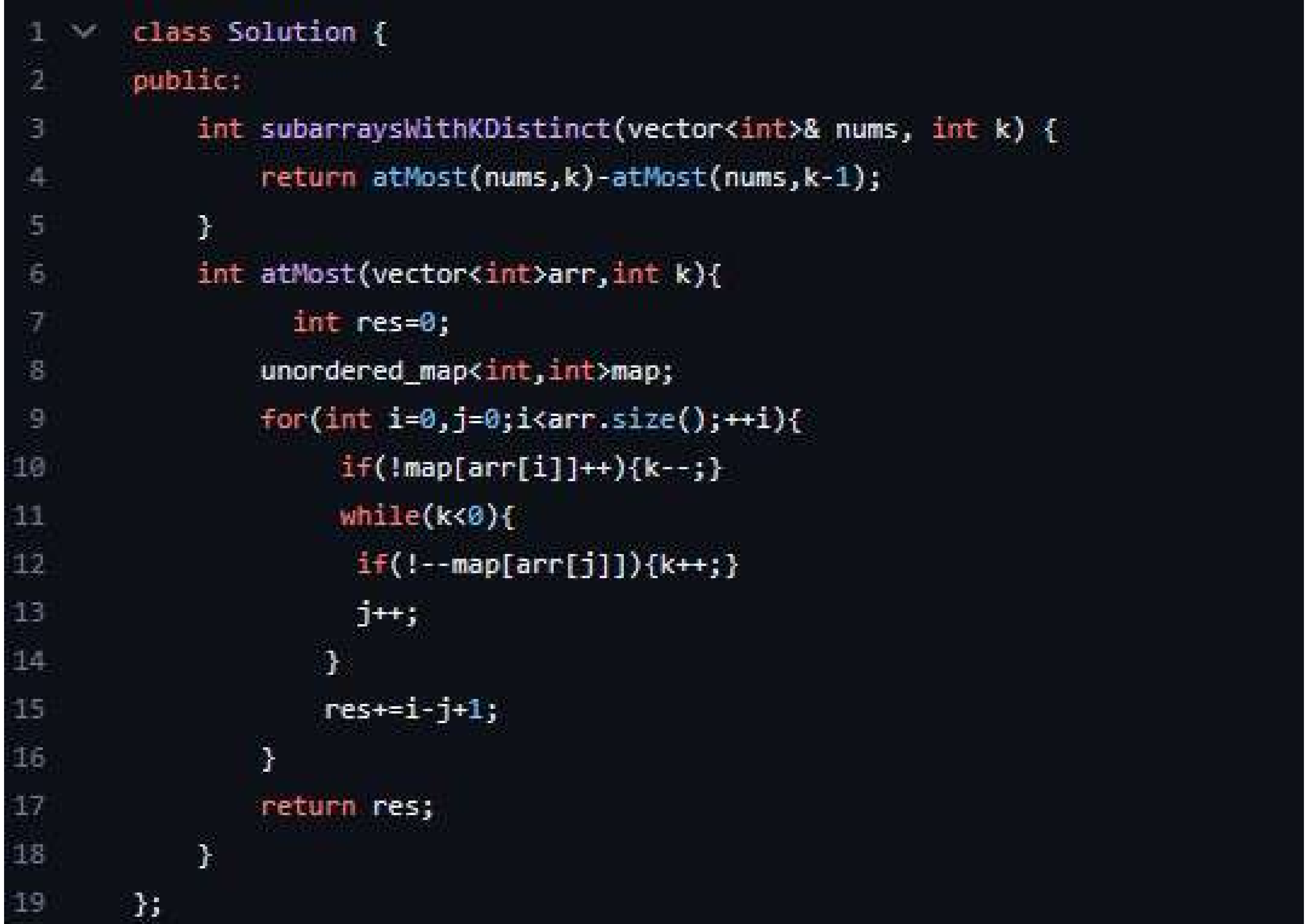
```
Input: nums = [1,2,1,2,3], k = 2
Output: 7
Explanation: Subarrays formed with exactly 2 different integers: [1,2], [2,1], [1,2], [2,3], [1,2,1], [2,1,2], [1,2,1,2]
```

Example 2:

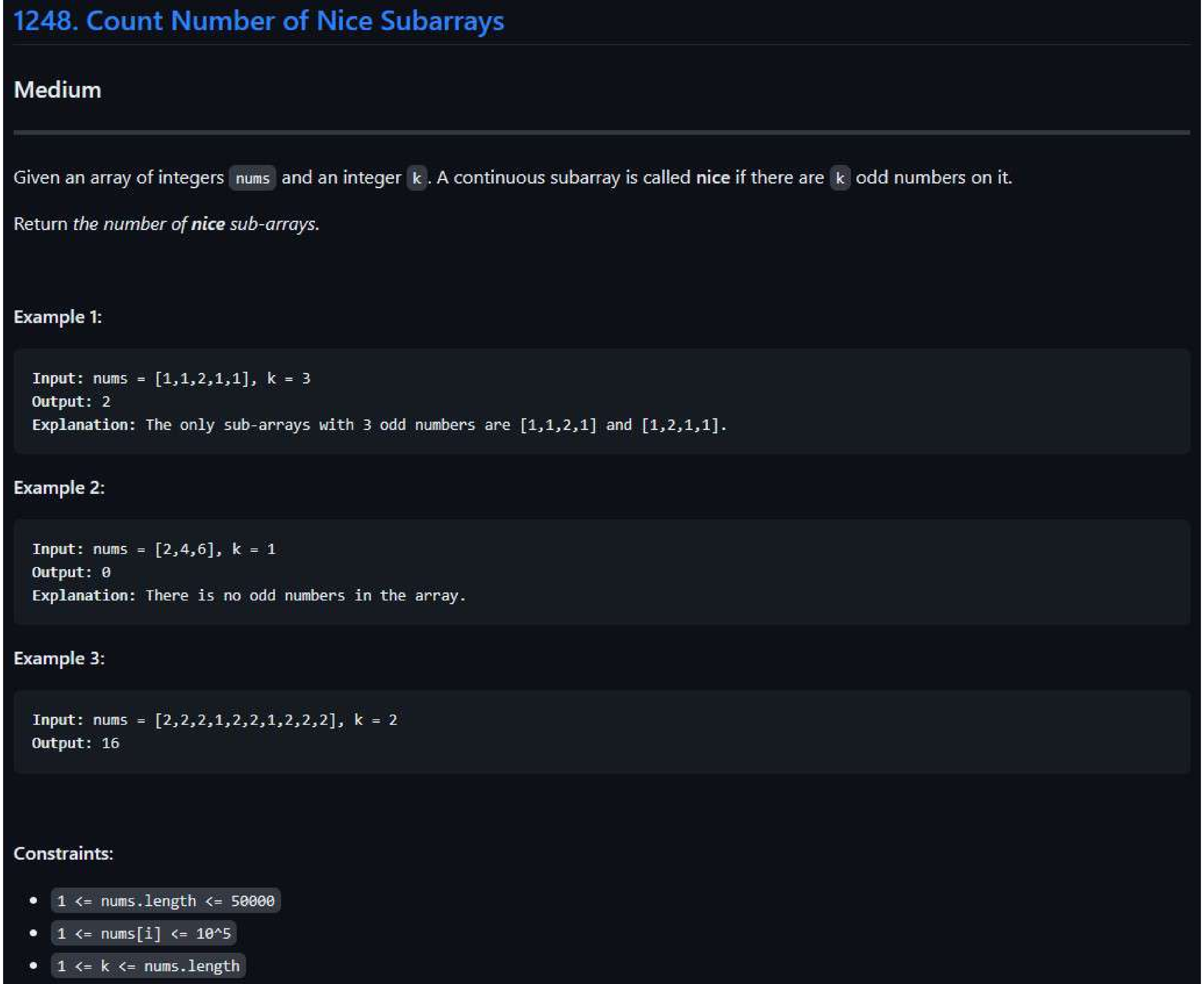
```
Input: nums = [1,2,1,3,4], k = 3
Output: 3
Explanation: Subarrays formed with exactly 3 different integers: [1,2,1,3], [2,1,3], [1,3,4].
```

Constraints:

- `1 <= nums.length <= 2 * ` 10^4
- `1 <= nums[i], k <= nums.length`



```cpp
class Solution {
public:
    int subarraysWithKDistinct(vector<int>& nums, int k) {
        return atMost(nums,k)-atMost(nums,k-1);
    }
    int atMost(vector<int>arr,int k){
          int res=0;
        unordered_map<int,int>map;
        for(int i=0,j=0;i<arr.size();++i){
            if(!map[arr[i]]++){k--;}
            while(k<0){
             if(!--map[arr[j]]){k++;}
             j++;
           }
          res+=i-j+1;
        }
        return res;
    }
};
```



1248. Count Number of Nice Subarrays

Medium

Given an array of integers `nums` and an integer `k`. A continuous subarray is called **nice** if there are `k` odd numbers on it.

Return *the number of **nice** sub-arrays.*

Example 1:

```
Input: nums = [1,1,2,1,1], k = 3
Output: 2
Explanation: The only sub-arrays with 3 odd numbers are [1,1,2,1] and [1,2,1,1].
```

Example 2:

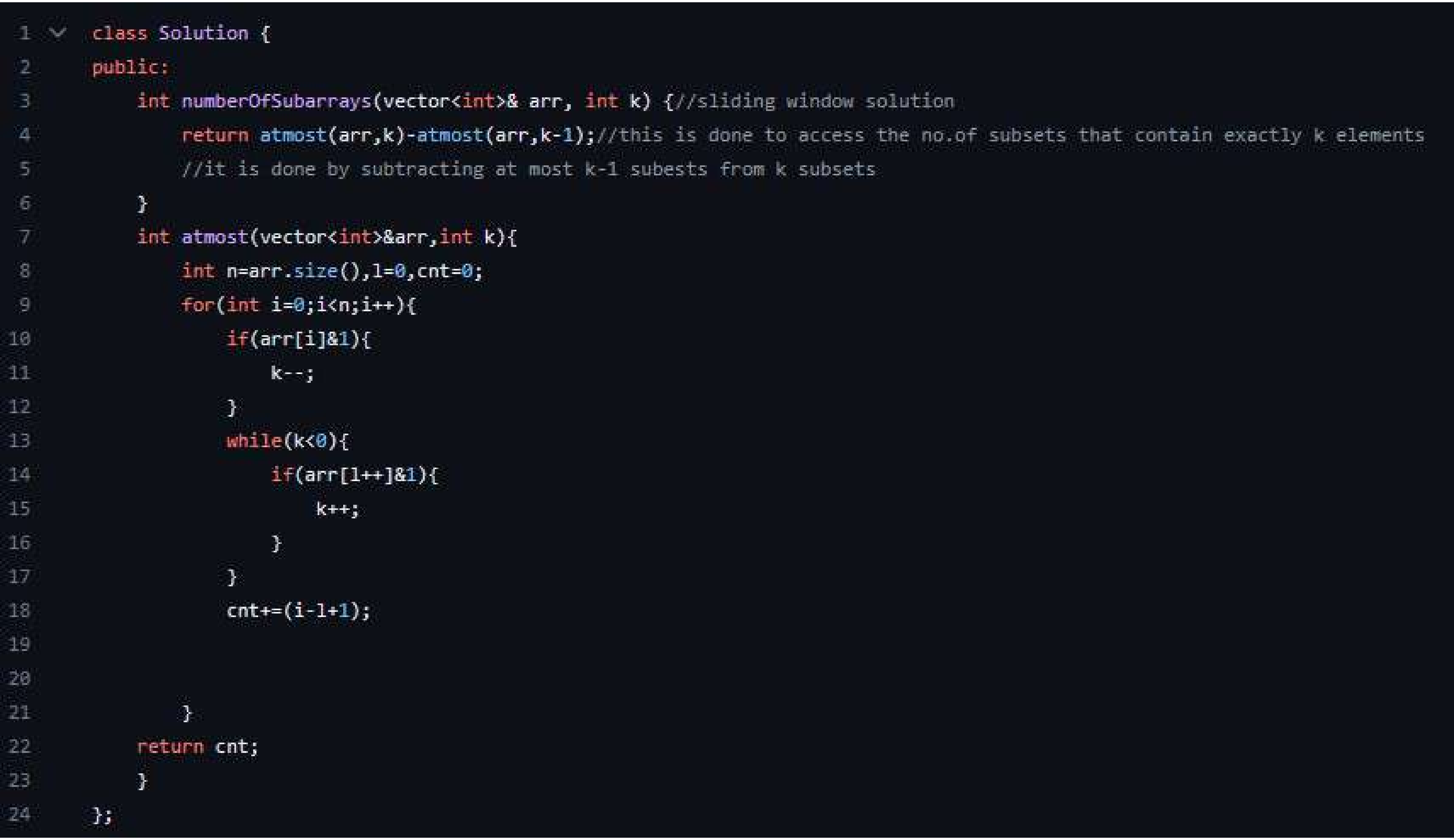
```
Input: nums = [2,4,6], k = 1
Output: 0
Explanation: There is no odd numbers in the array.
```

Example 3:

```
Input: nums = [2,2,2,1,2,2,1,2,2,2], k = 2
Output: 16
```

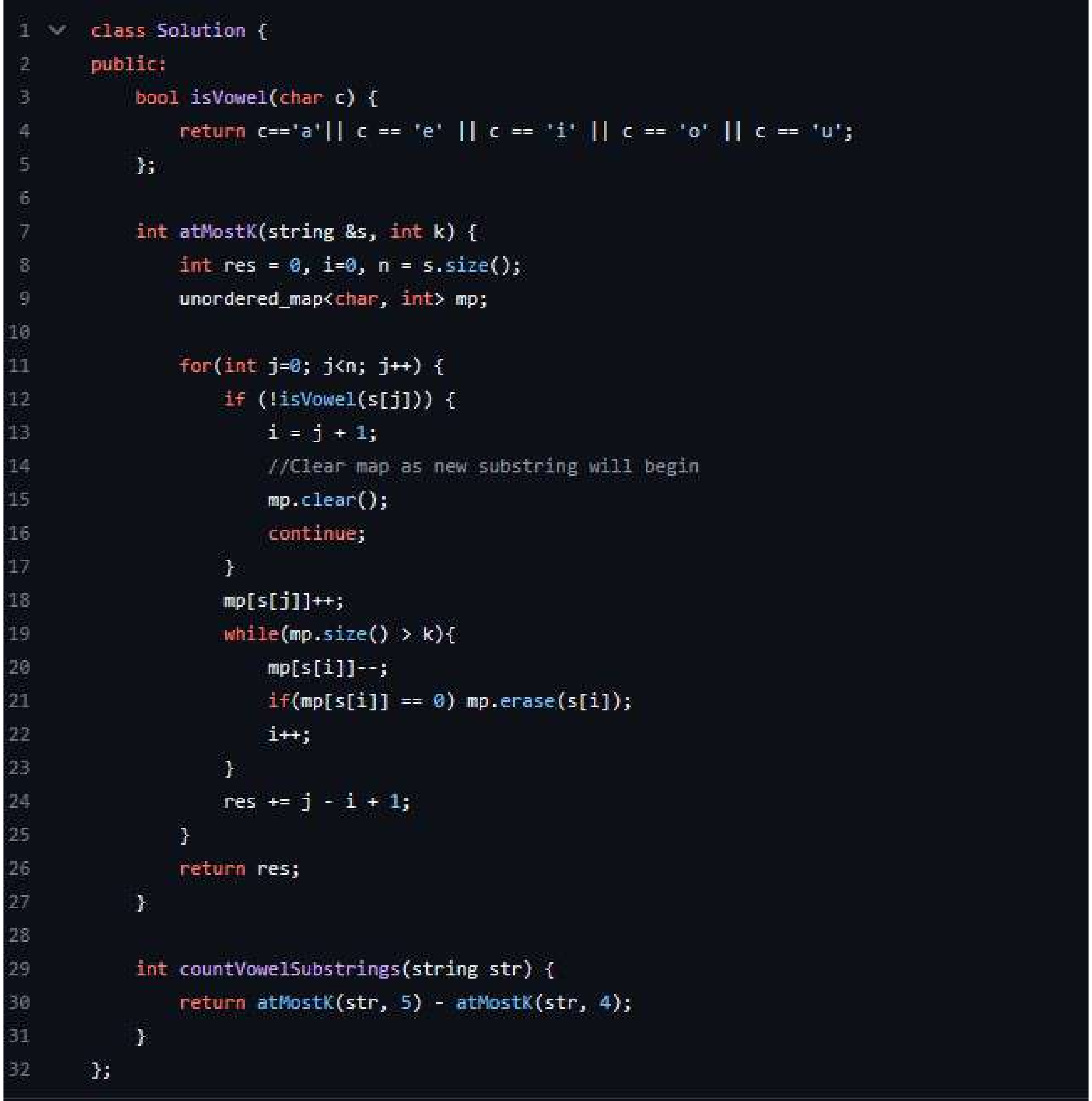
Constraints:

- `1 <= nums.length <= 50000`
- `1 <= nums[i] <= 10^5`
- `1 <= k <= nums.length`



```
class Solution {
public:
    int numberOfSubarrays(vector<int>& arr, int k) {//sliding window solution
        return atmost(arr,k)-atmost(arr,k-1);//this is done to access the no.of subsets that contain exactly k elements
        //it is done by subtracting at most k-1 subests from k subsets
    }
    int atmost(vector<int>&arr,int k){
        int n=arr.size(),l=0,cnt=0;
        for(int i=0;i<n;i++){
            if(arr[i]&1){
                k--;
            }
            while(k<0){
                if(arr[l++]&1){
                    k++;
                }
            }
            cnt+=(i-l+1);


        }
    return cnt;
    }
};
```

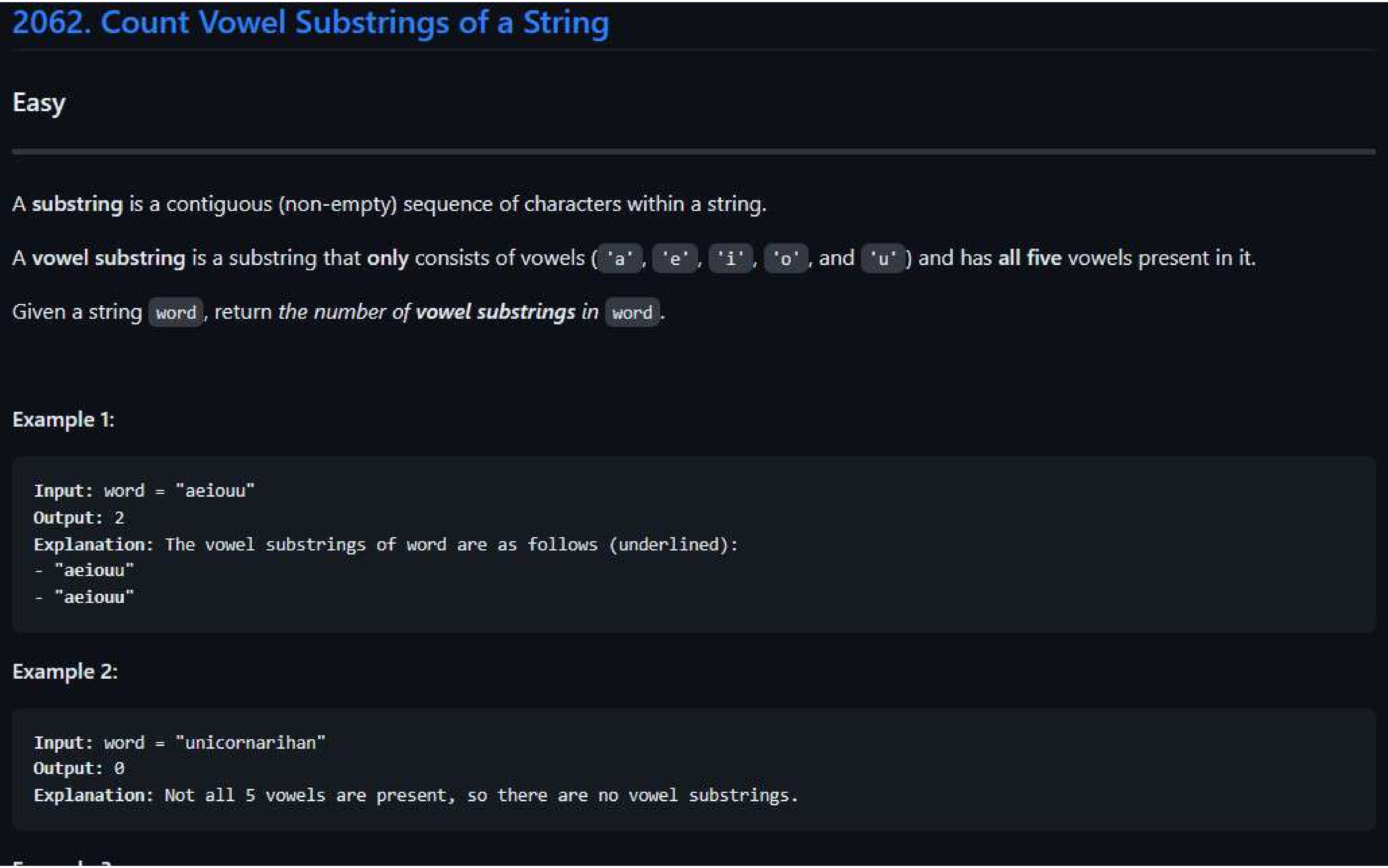


```cpp
class Solution {
public:
    bool isVowel(char c) {
        return c=='a'|| c == 'e' || c == 'i' || c == 'o' || c == 'u';
    };

    int atMostK(string &s, int k) {
        int res = 0, i=0, n = s.size();
        unordered_map<char, int> mp;

        for(int j=0; j<n; j++) {
            if (!isVowel(s[j])) {
                i = j + 1;
                //Clear map as new substring will begin
                mp.clear();
                continue;
            }
            mp[s[j]]++;
            while(mp.size() > k){
                mp[s[i]]--;
                if(mp[s[i]] == 0) mp.erase(s[i]);
                i++;
            }
            res += j - i + 1;
        }
        return res;
    }

    int countVowelSubstrings(string str) {
        return atMostK(str, 5) - atMostK(str, 4);
    }
};
```



2062. Count Vowel Substrings of a String

Easy

A **substring** is a contiguous (non-empty) sequence of characters within a string.

A **vowel substring** is a substring that **only** consists of vowels (`'a'`, `'e'`, `'i'`, `'o'`, and `'u'`) and has **all five** vowels present in it.

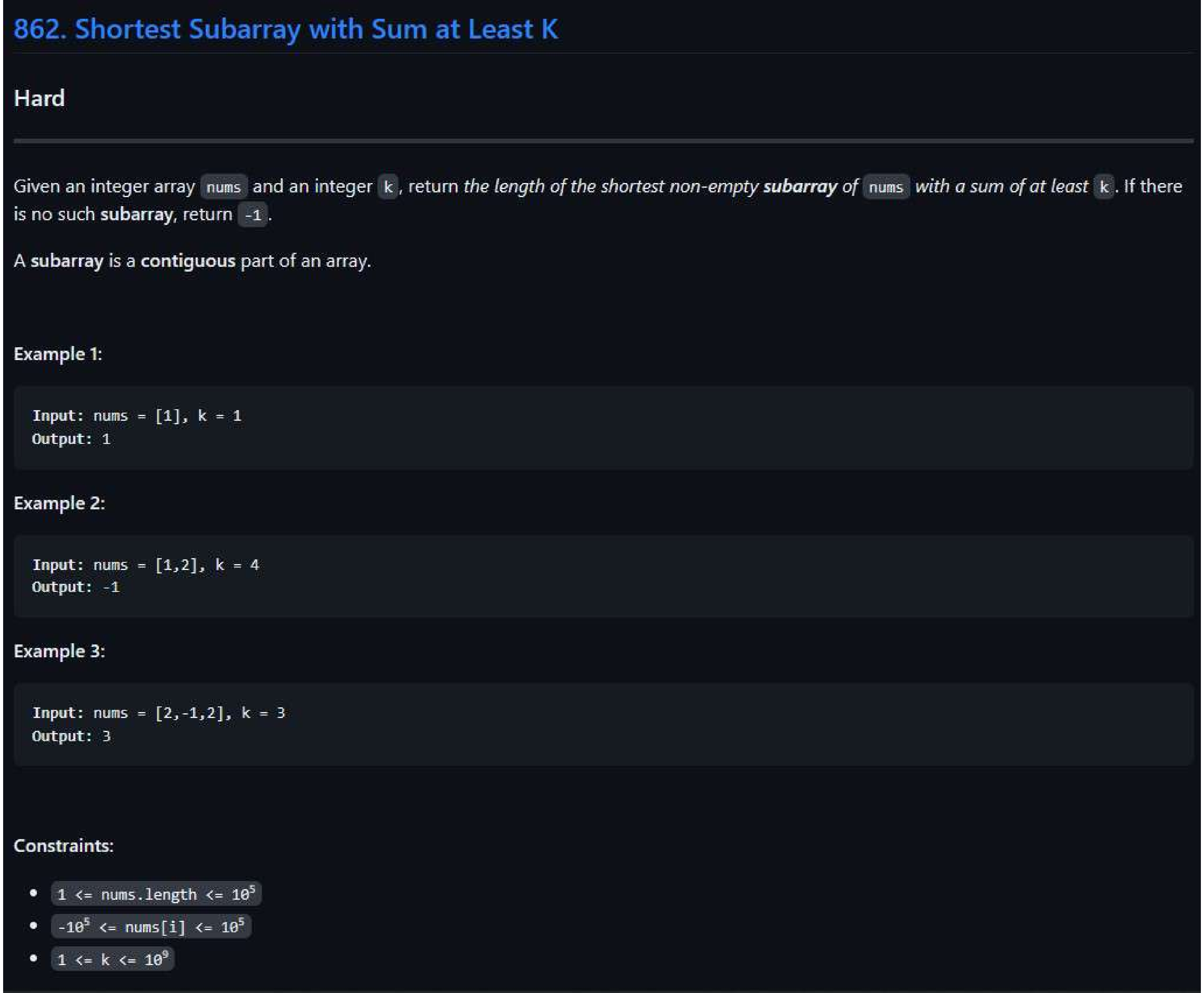
Given a string `word`, return *the number of **vowel substrings** in* `word`.

Example 1:

```
Input: word = "aeiouu"
Output: 2
Explanation: The vowel substrings of word are as follows (underlined):
- "aeiouu"
- "aeiouu"
```

Example 2:

```
Input: word = "unicornarihan"
Output: 0
Explanation: Not all 5 vowels are present, so there are no vowel substrings.
```



862. Shortest Subarray with Sum at Least K

Hard

Given an integer array `nums` and an integer `k`, return *the length of the shortest non-empty* ***subarray*** *of* `nums` *with a sum of at least* `k`. If there is no such **subarray**, return `-1`.

A **subarray** is a **contiguous** part of an array.

Example 1:

```
Input: nums = [1], k = 1
Output: 1
```

Example 2:

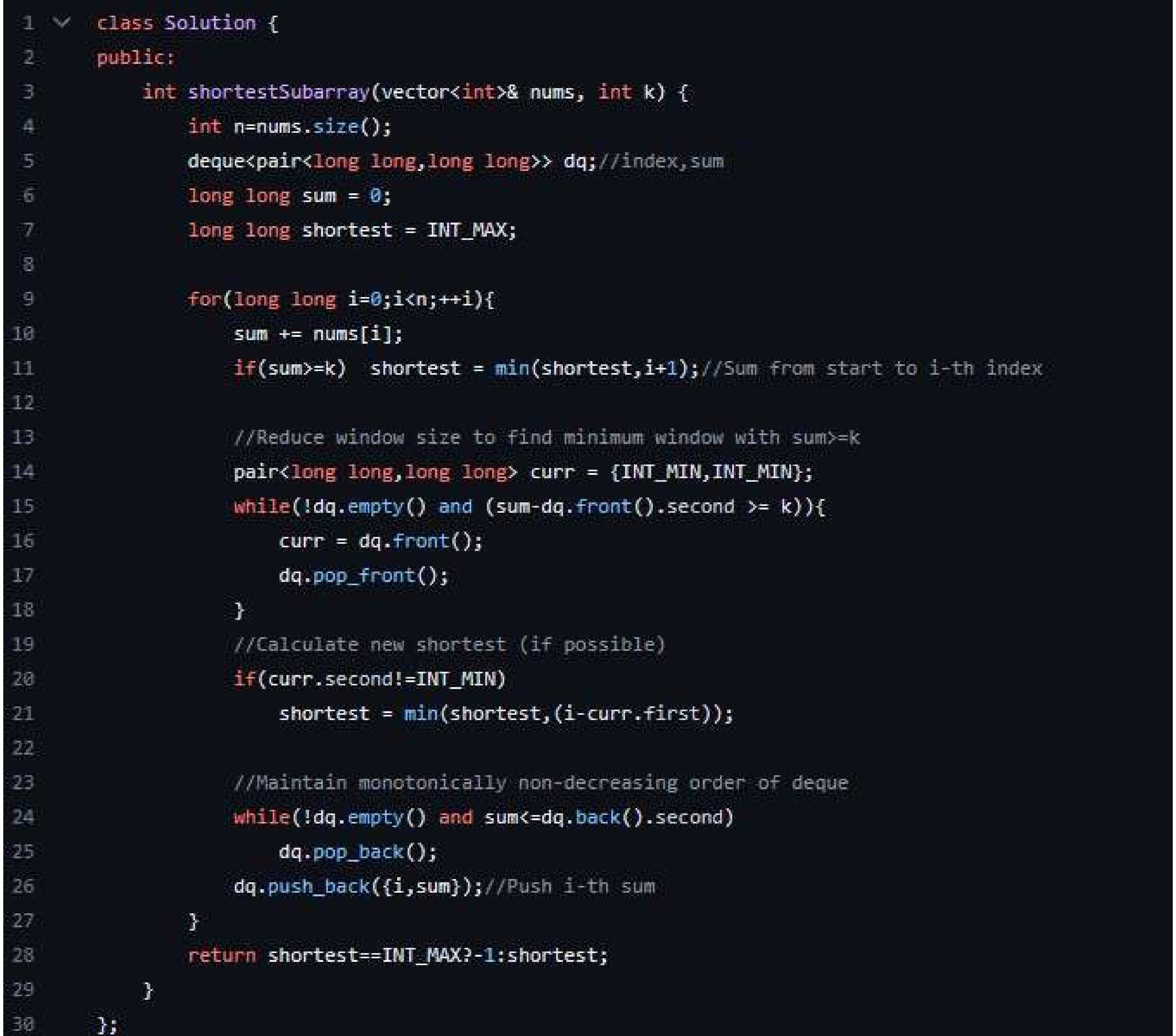
```
Input: nums = [1,2], k = 4
Output: -1
```

Example 3:

```
Input: nums = [2,-1,2], k = 3
Output: 3
```

Constraints:

- `1 <= nums.length <=` 10^5
- -10^5 `<= nums[i] <=` 10^5
- `1 <= k <=` 10^9

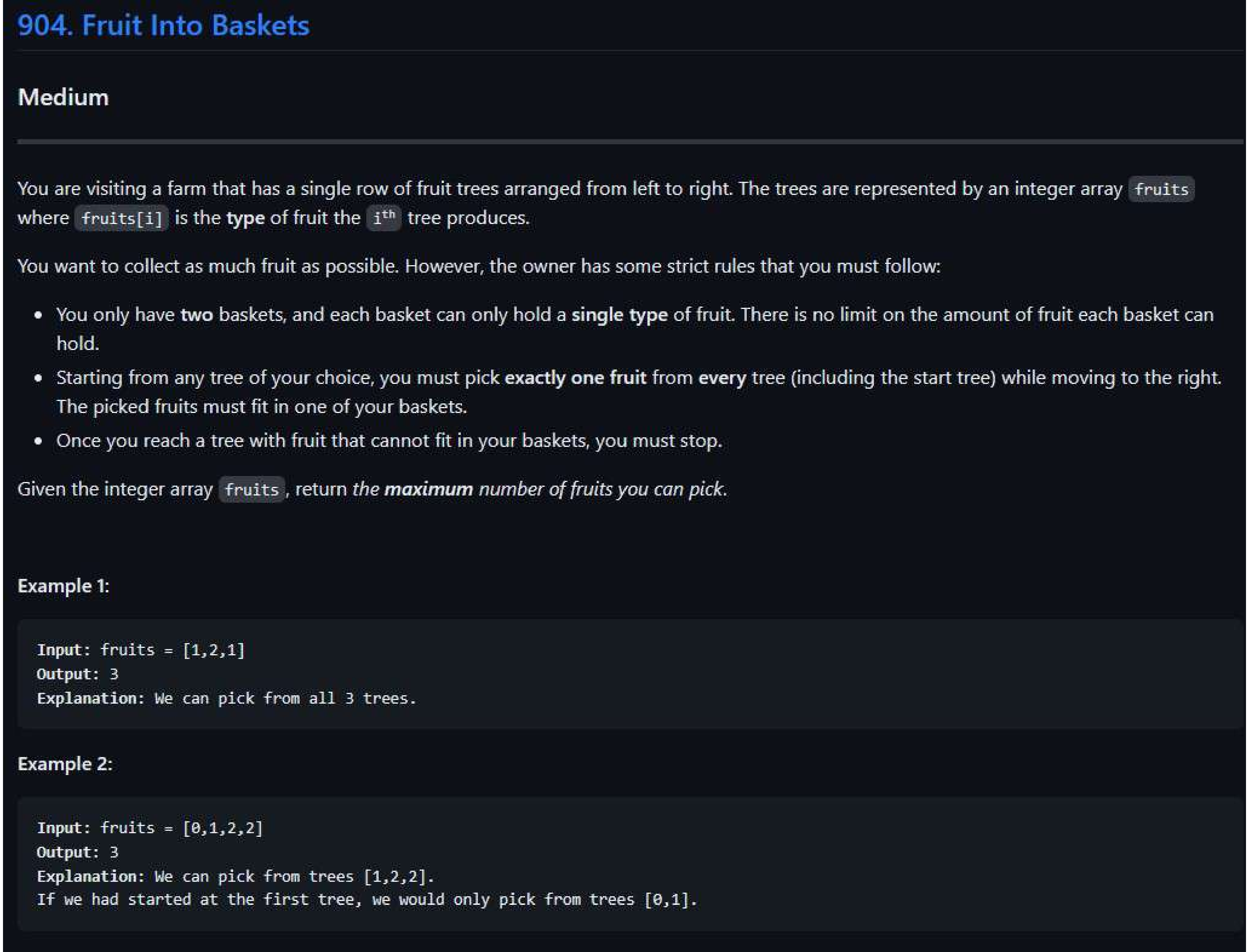


```cpp
class Solution {
public:
    int shortestSubarray(vector<int>& nums, int k) {
        int n=nums.size();
        deque<pair<long long,long long>> dq;//index,sum
        long long sum = 0;
        long long shortest = INT_MAX;

        for(long long i=0;i<n;++i){
            sum += nums[i];
            if(sum>=k)  shortest = min(shortest,i+1);//Sum from start to i-th index

            //Reduce window size to find minimum window with sum>=k
            pair<long long,long long> curr = {INT_MIN,INT_MIN};
            while(!dq.empty() and (sum-dq.front().second >= k)){
                curr = dq.front();
                dq.pop_front();
            }
            //Calculate new shortest (if possible)
            if(curr.second!=INT_MIN)
                shortest = min(shortest,(i-curr.first));

            //Maintain monotonically non-decreasing order of deque
            while(!dq.empty() and sum<=dq.back().second)
                dq.pop_back();
            dq.push_back({i,sum});//Push i-th sum
        }
        return shortest==INT_MAX?-1:shortest;
    }
};
```



904. Fruit Into Baskets

Medium

You are visiting a farm that has a single row of fruit trees arranged from left to right. The trees are represented by an integer array `fruits` where `fruits[i]` is the **type** of fruit the `i^th` tree produces.

You want to collect as much fruit as possible. However, the owner has some strict rules that you must follow:

- You only have **two** baskets, and each basket can only hold a **single type** of fruit. There is no limit on the amount of fruit each basket can hold.
- Starting from any tree of your choice, you must pick **exactly one fruit** from **every** tree (including the start tree) while moving to the right. The picked fruits must fit in one of your baskets.
- Once you reach a tree with fruit that cannot fit in your baskets, you must stop.

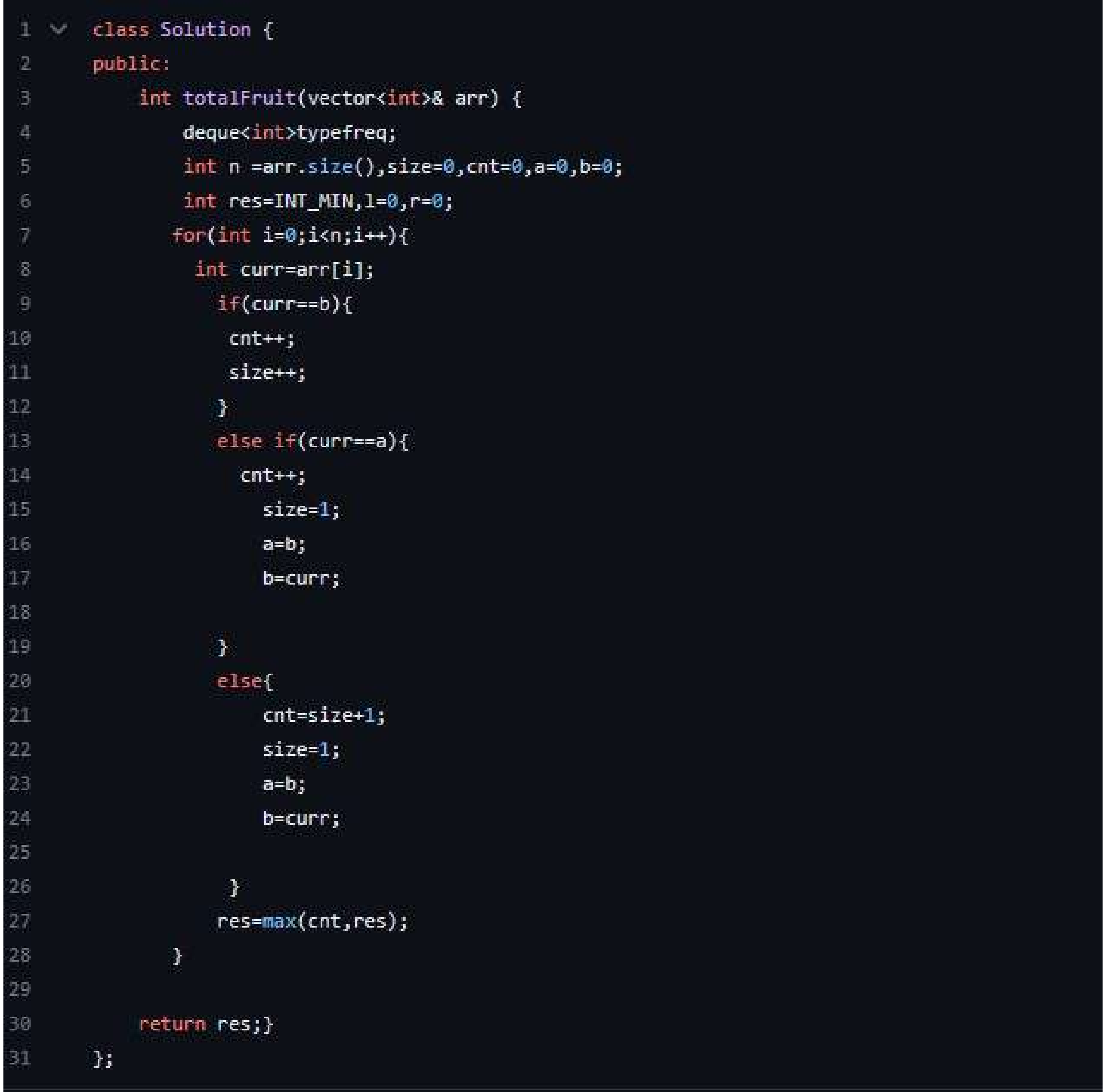
Given the integer array `fruits`, return *the **maximum** number of fruits you can pick*.

Example 1:

```
Input: fruits = [1,2,1]
Output: 3
Explanation: We can pick from all 3 trees.
```

Example 2:

```
Input: fruits = [0,1,2,2]
Output: 3
Explanation: We can pick from trees [1,2,2].
If we had started at the first tree, we would only pick from trees [0,1].
```



```cpp
class Solution {
public:
    int totalFruit(vector<int>& arr) {
        deque<int>typefreq;
        int n =arr.size(),size=0,cnt=0,a=0,b=0;
        int res=INT_MIN,l=0,r=0;
       for(int i=0;i<n;i++){
         int curr=arr[i];
           if(curr==b){
            cnt++;
            size++;
           }
           else if(curr==a){
             cnt++;
               size=1;
               a=b;
               b=curr;

           }
           else{
               cnt=size+1;
               size=1;
               a=b;
               b=curr;

            }
           res=max(cnt,res);
       }

    return res;}
};
```

930. Binary Subarrays With Sum

Medium

Given a binary array `nums` and an integer `goal`, return *the number of non-empty* ***subarrays*** *with a sum* `goal`.

A **subarray** is a contiguous part of the array.

Example 1:

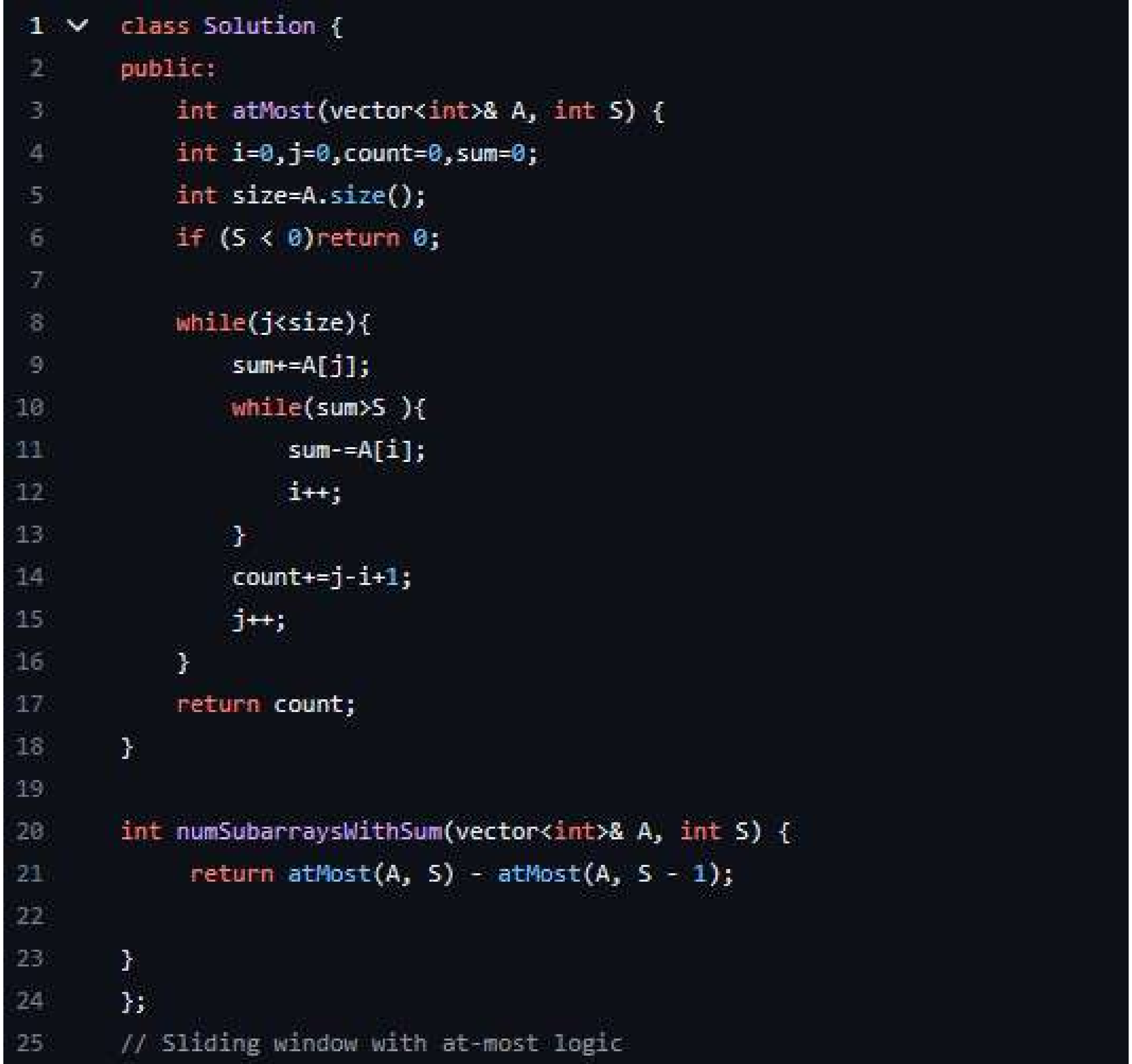
```
Input: nums = [1,0,1,0,1], goal = 2
Output: 4
Explanation: The 4 subarrays are bolded and underlined below:
[1,0,1,0,1]
[1,0,1,0,1]
[1,0,1,0,1]
[1,0,1,0,1]
```

Example 2:

```
Input: nums = [0,0,0,0,0], goal = 0
Output: 15
```

Constraints:

- `1 <= nums.length <= 3 * 10`^{4}
- `nums[i]` is either `0` or `1`.
- `0 <= goal <= nums.length`

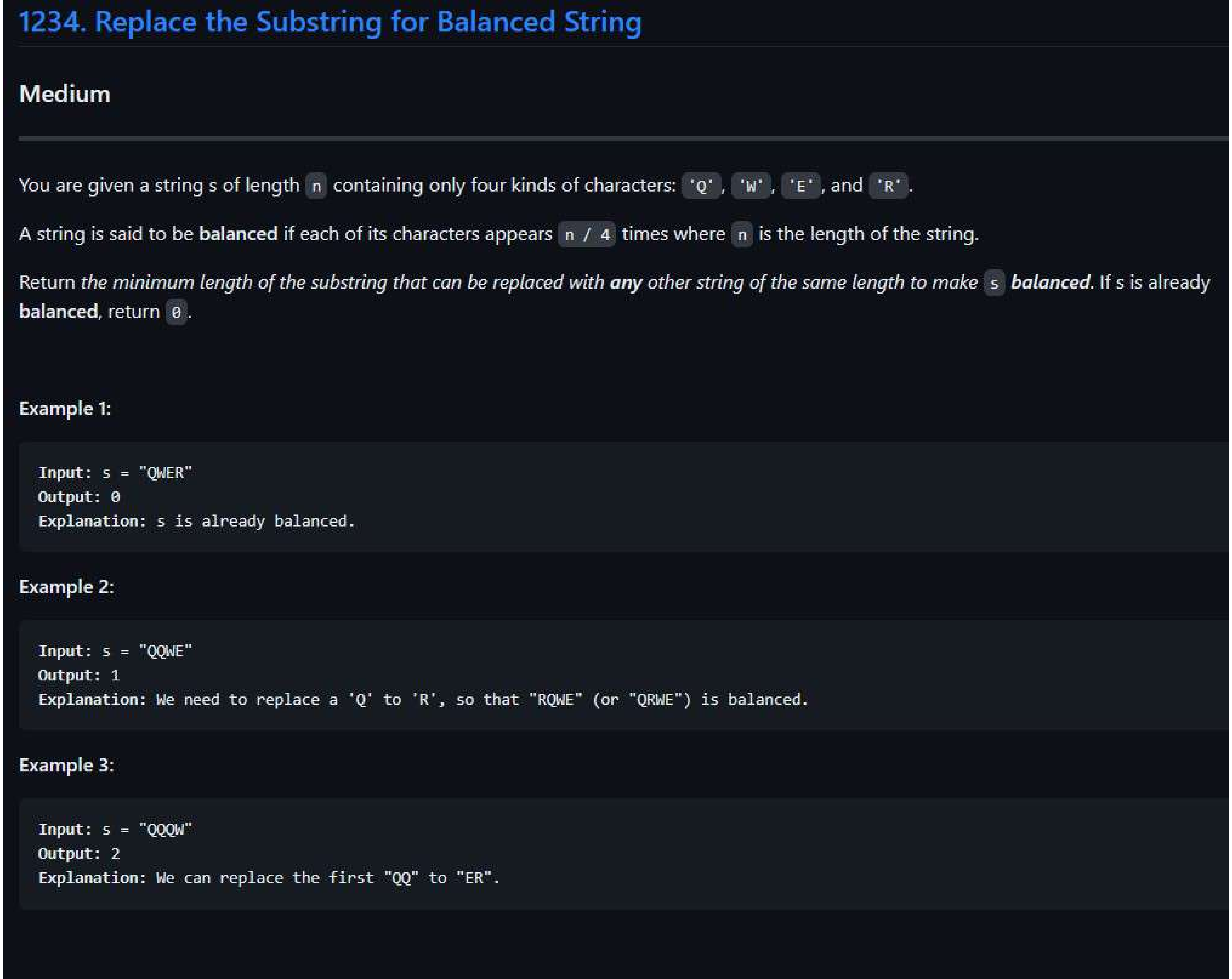


```cpp
class Solution {
public:
    int atMost(vector<int>& A, int S) {
    int i=0,j=0,count=0,sum=0;
    int size=A.size();
    if (S < 0)return 0;

    while(j<size){
        sum+=A[j];
        while(sum>S ){
            sum-=A[i];
            i++;
        }
        count+=j-i+1;
        j++;
    }
    return count;
}

int numSubarraysWithSum(vector<int>& A, int S) {
     return atMost(A, S) - atMost(A, S - 1);

}
};
// Sliding window with at-most logic
```



1234. Replace the Substring for Balanced String

Medium

You are given a string s of length `n` containing only four kinds of characters: `'Q'`, `'W'`, `'E'`, and `'R'`.

A string is said to be **balanced** if each of its characters appears `n / 4` times where `n` is the length of the string.

Return *the minimum length of the substring that can be replaced with **any** other string of the same length to make* `s` ***balanced***. If s is already **balanced**, return `0`.

Example 1:

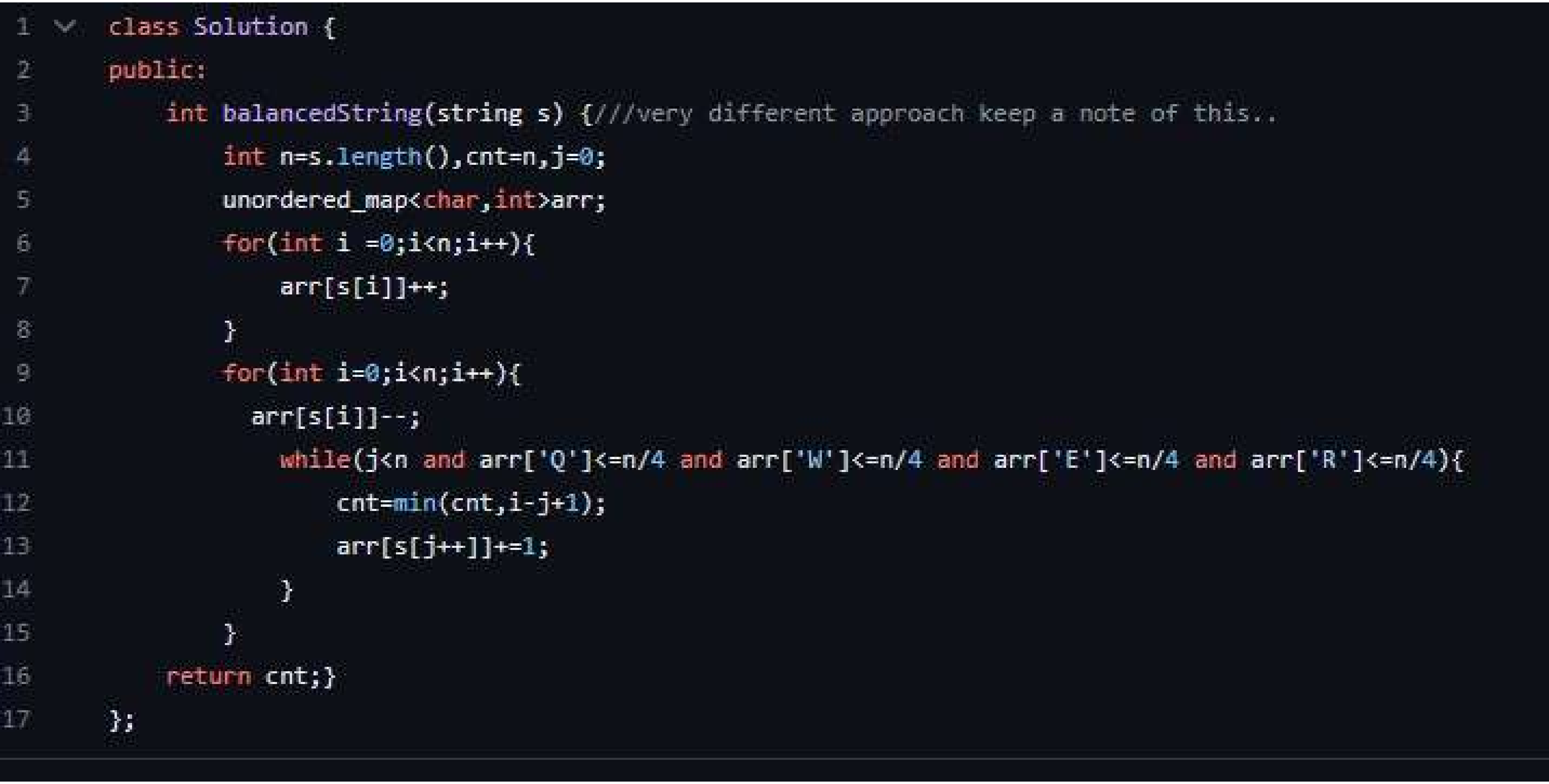
```
Input: s = "QWER"
Output: 0
Explanation: s is already balanced.
```

Example 2:

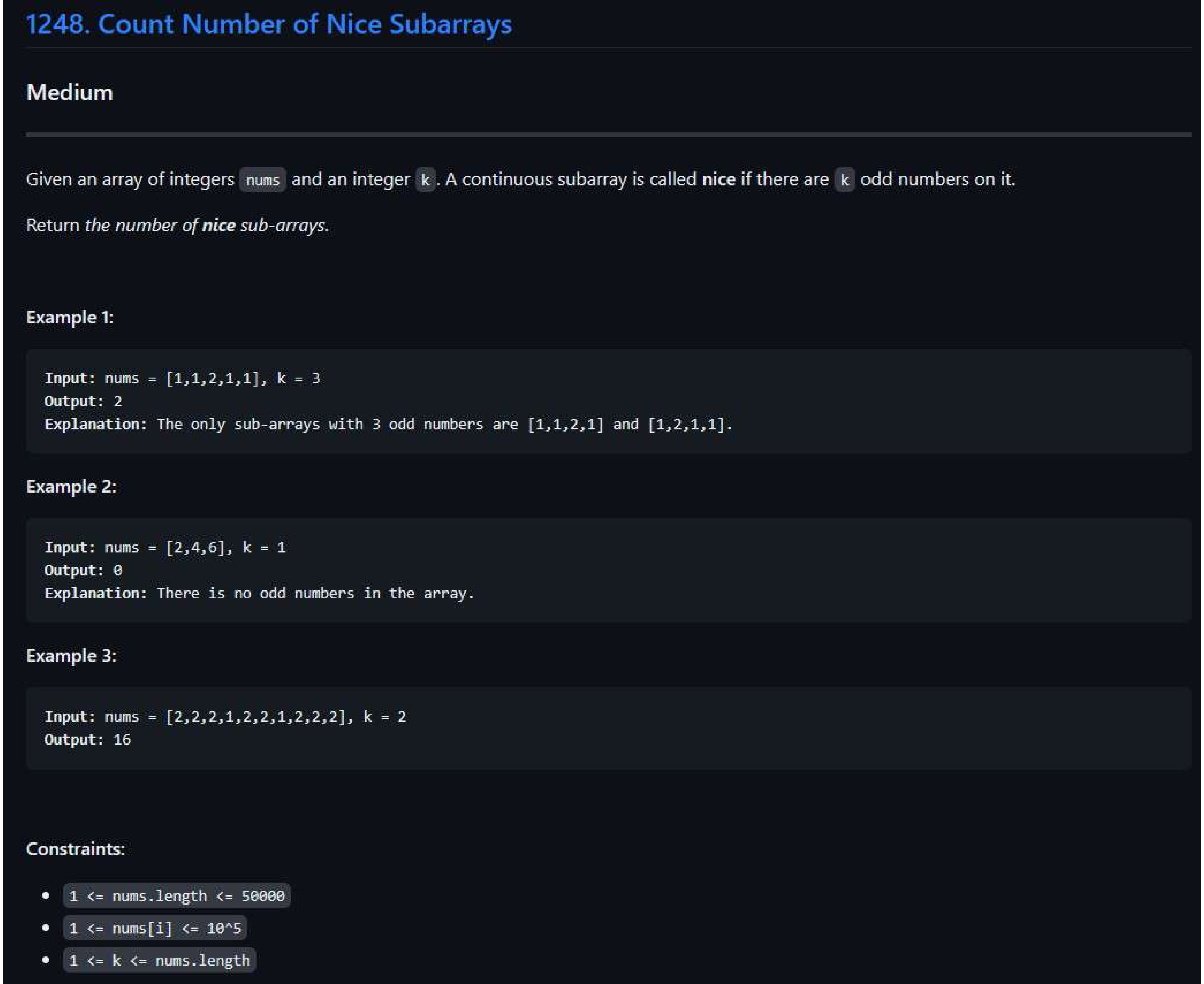
```
Input: s = "QQWE"
Output: 1
Explanation: We need to replace a 'Q' to 'R', so that "RQWE" (or "QRWE") is balanced.
```

Example 3:

```
Input: s = "QQQW"
Output: 2
Explanation: We can replace the first "QQ" to "ER".
```



```cpp
class Solution {
public:
    int balancedString(string s) {///very different approach keep a note of this..
        int n=s.length(),cnt=n,j=0;
        unordered_map<char,int>arr;
        for(int i =0;i<n;i++){
            arr[s[i]]++;
        }
        for(int i=0;i<n;i++){
          arr[s[i]]--;
            while(j<n and arr['Q']<=n/4 and arr['W']<=n/4 and arr['E']<=n/4 and arr['R']<=n/4){
                cnt=min(cnt,i-j+1);
                arr[s[j++]]+=1;
            }
        }
    return cnt;}
};
```



1248. Count Number of Nice Subarrays

Medium

Given an array of integers `nums` and an integer `k`. A continuous subarray is called **nice** if there are `k` odd numbers on it.

Return *the number of **nice** sub-arrays*.

Example 1:

```
Input: nums = [1,1,2,1,1], k = 3
Output: 2
Explanation: The only sub-arrays with 3 odd numbers are [1,1,2,1] and [1,2,1,1].
```

Example 2:

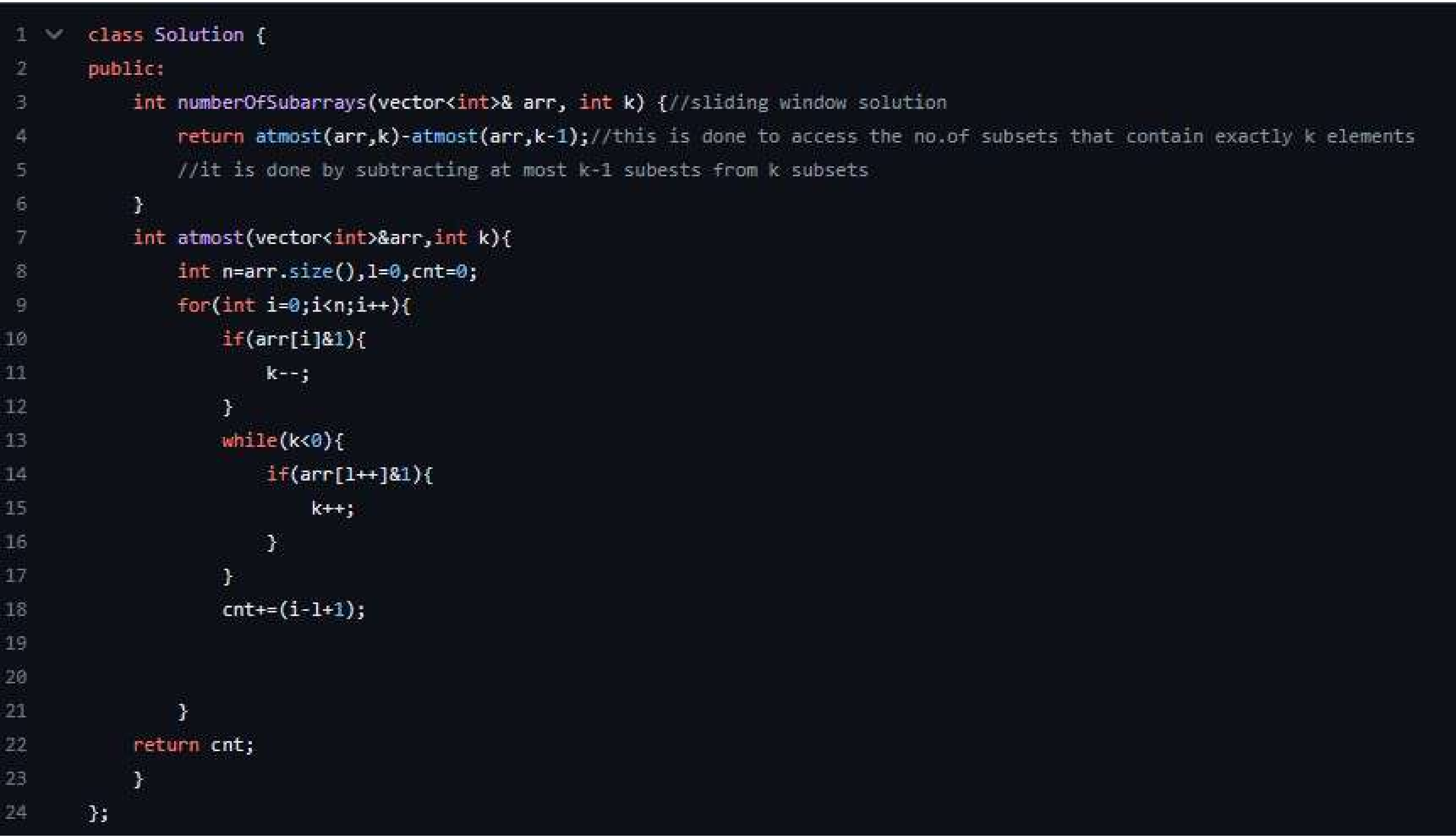
```
Input: nums = [2,4,6], k = 1
Output: 0
Explanation: There is no odd numbers in the array.
```

Example 3:

```
Input: nums = [2,2,2,1,2,2,1,2,2,2], k = 2
Output: 16
```

Constraints:

- `1 <= nums.length <= 50000`
- `1 <= nums[i] <= 10^5`
- `1 <= k <= nums.length`



```
class Solution {
public:
    int numberOfSubarrays(vector<int>& arr, int k) {//sliding window solution
        return atmost(arr,k)-atmost(arr,k-1);//this is done to access the no.of subsets that contain exactly k elements
        //it is done by subtracting at most k-1 subests from k subsets
    }
    int atmost(vector<int>&arr,int k){
        int n=arr.size(),l=0,cnt=0;
        for(int i=0;i<n;i++){
            if(arr[i]&1){
                k--;
            }
            if(k<0){
                if(arr[l++]&1){
                    k++;
                }
            }
            cnt+=(i-l+1);


        }
    return cnt;
    }
};
```

1358. Number of Substrings Containing All Three Characters

Medium

Given a string `s` consisting only of characters *a*, *b* and *c*.

Return the number of substrings containing **at least** one occurrence of all these characters *a*, *b* and *c*.

Example 1:

```
Input: s = "abcabc"
Output: 10
Explanation: The substrings containing at least one occurrence of the characters a, b and c are "abc", "abca", "abcab", "abcabc", "bc
```

Example 2:

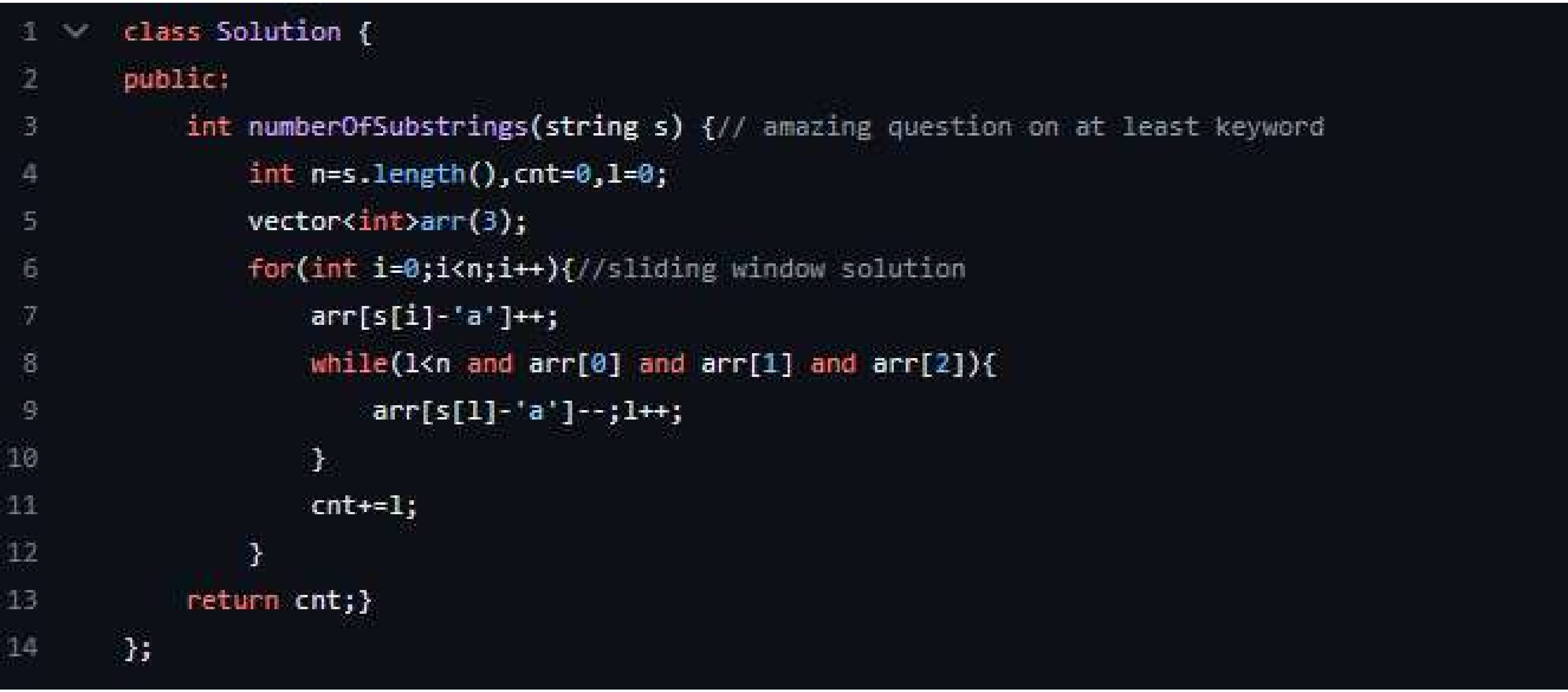
```
Input: s = "aaacb"
Output: 3
Explanation: The substrings containing at least one occurrence of the characters a, b and c are "aaacb", "aacb" and "acb".
```

Example 3:

```
Input: s = "abc"
Output: 1
```

Constraints:

- `3 <= s.length <= 5 x 10^4`
- `s` only consists of *a*, *b* or *c* characters.



```
class Solution {
public:
    int numberOfSubstrings(string s) {// amazing question on at least keyword
        int n=s.length(),cnt=0,l=0;
        vector<int>arr(3);
        for(int i=0;i<n;i++){//sliding window solution
            arr[s[i]-'a']++;
            while(l<n and arr[0] and arr[1] and arr[2]){
                arr[s[l]-'a']--;l++;
            }
            cnt+=l;
        }
    return cnt;}
};
```

1438. Longest Continuous Subarray With Absolute Diff Less Than or Equal to Limit

Medium

Given an array of integers `nums` and an integer `limit`, return the size of the longest **non-empty** subarray such that the absolute difference between any two elements of this subarray is less than or equal to `limit`.

Example 1:

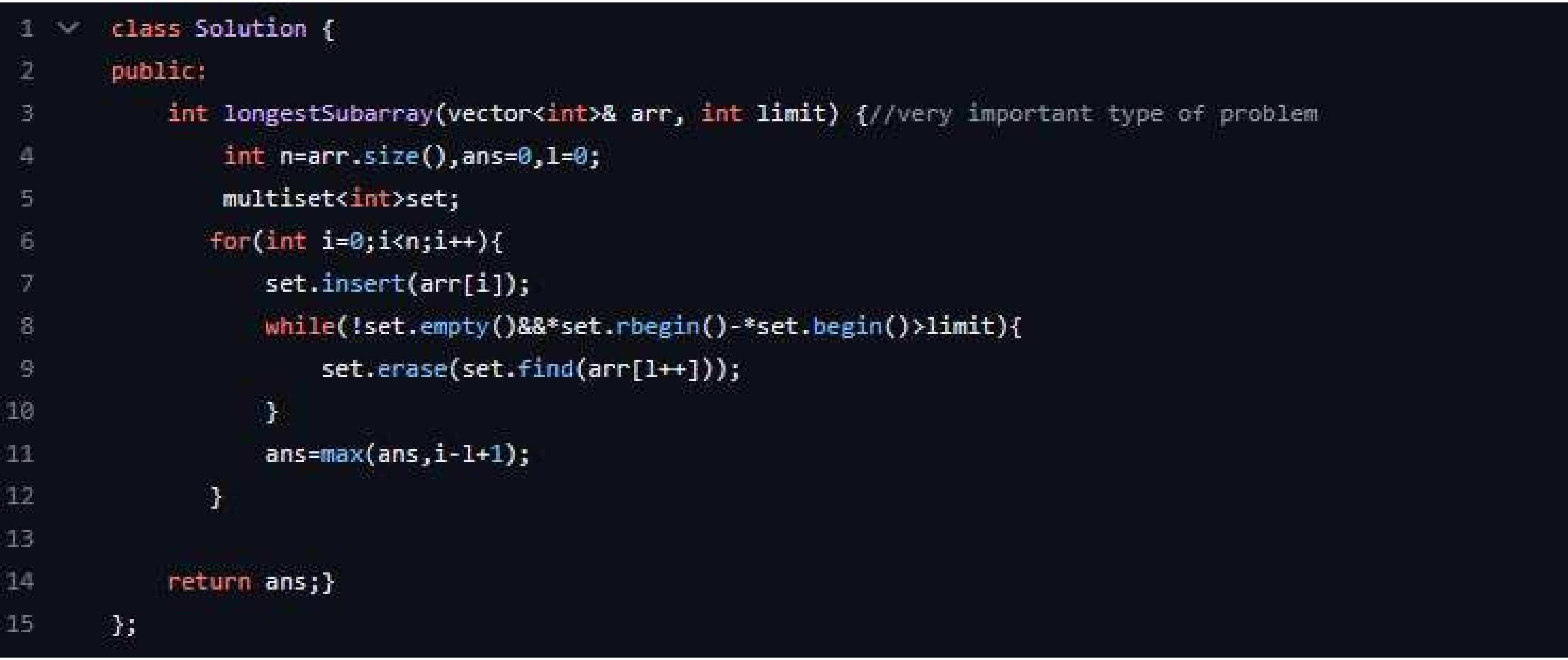
```
Input: nums = [8,2,4,7], limit = 4
Output: 2
Explanation: All subarrays are:
[8] with maximum absolute diff |8-8| = 0 <= 4.
[8,2] with maximum absolute diff |8-2| = 6 > 4.
[8,2,4] with maximum absolute diff |8-2| = 6 > 4.
[8,2,4,7] with maximum absolute diff |8-2| = 6 > 4.
[2] with maximum absolute diff |2-2| = 0 <= 4.
[2,4] with maximum absolute diff |2-4| = 2 <= 4.
[2,4,7] with maximum absolute diff |2-7| = 5 > 4.
[4] with maximum absolute diff |4-4| = 0 <= 4.
[4,7] with maximum absolute diff |4-7| = 3 <= 4.
[7] with maximum absolute diff |7-7| = 0 <= 4.
Therefore, the size of the longest subarray is 2.
```

Example 2:

```
Input: nums = [10,1,2,4,7,2], limit = 5
Output: 4
Explanation: The subarray [2,4,7,2] is the longest since the maximum absolute diff is |2-7| = 5 <= 5.
```

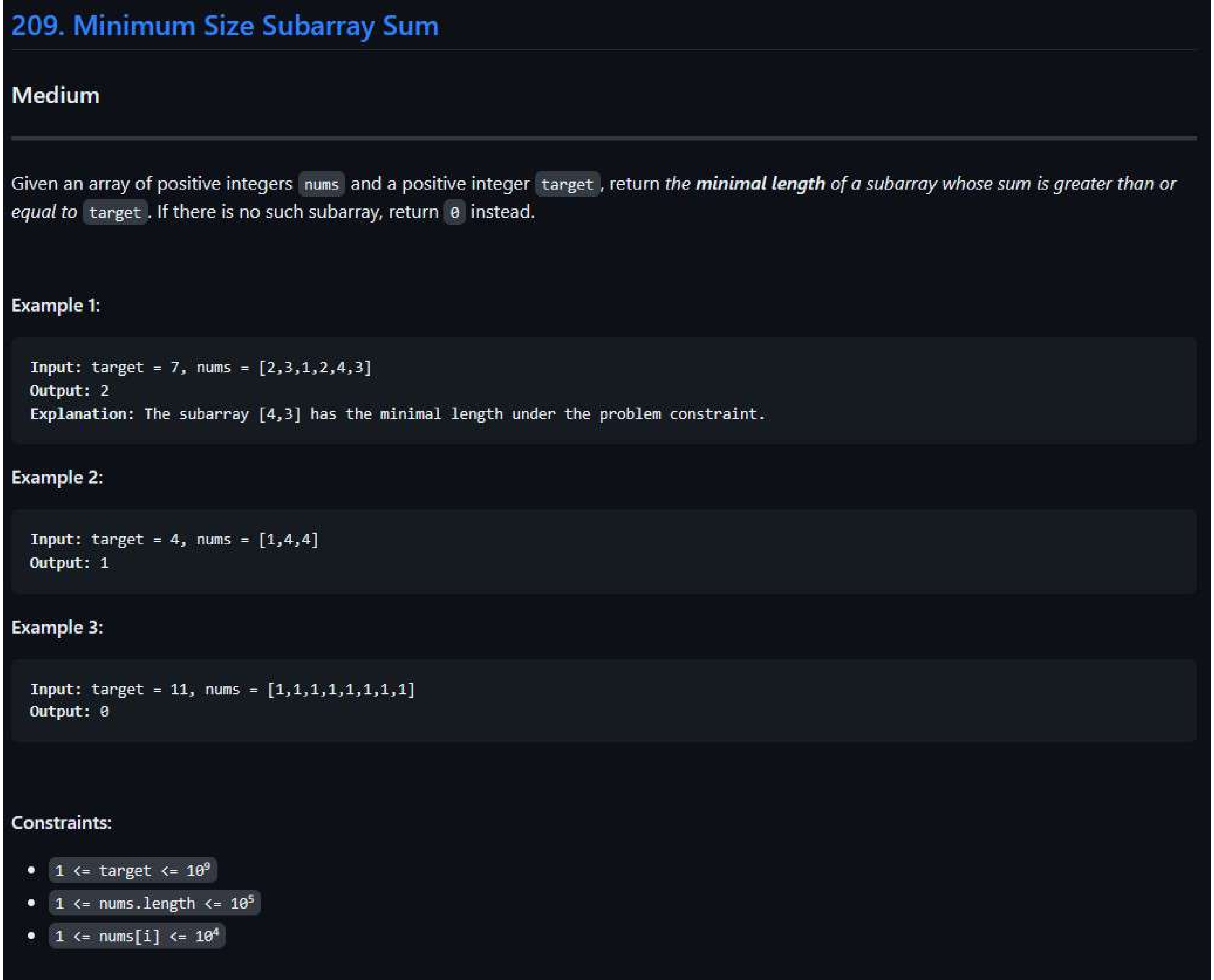
Example 3:

```
Input: nums = [4,2,2,2,4,4,2,2], limit = 0
Output: 3
```



```cpp
class Solution {
public:
    int longestSubarray(vector<int>& arr, int limit) {//very important type of problem
        int n=arr.size(),ans=0,l=0;
        multiset<int>set;
       for(int i=0;i<n;i++){
           set.insert(arr[i]);
           while(!set.empty()&&*set.rbegin()-*set.begin()>limit){
               set.erase(set.find(arr[l++]));
           }
           ans=max(ans,i-l+1);
       }

    return ans;}
};
```



209. Minimum Size Subarray Sum

Medium

Given an array of positive integers `nums` and a positive integer `target`, return *the **minimal length** of a subarray whose sum is greater than or equal to* `target`. If there is no such subarray, return `0` instead.

Example 1:

```
Input: target = 7, nums = [2,3,1,2,4,3]
Output: 2
Explanation: The subarray [4,3] has the minimal length under the problem constraint.
```

Example 2:

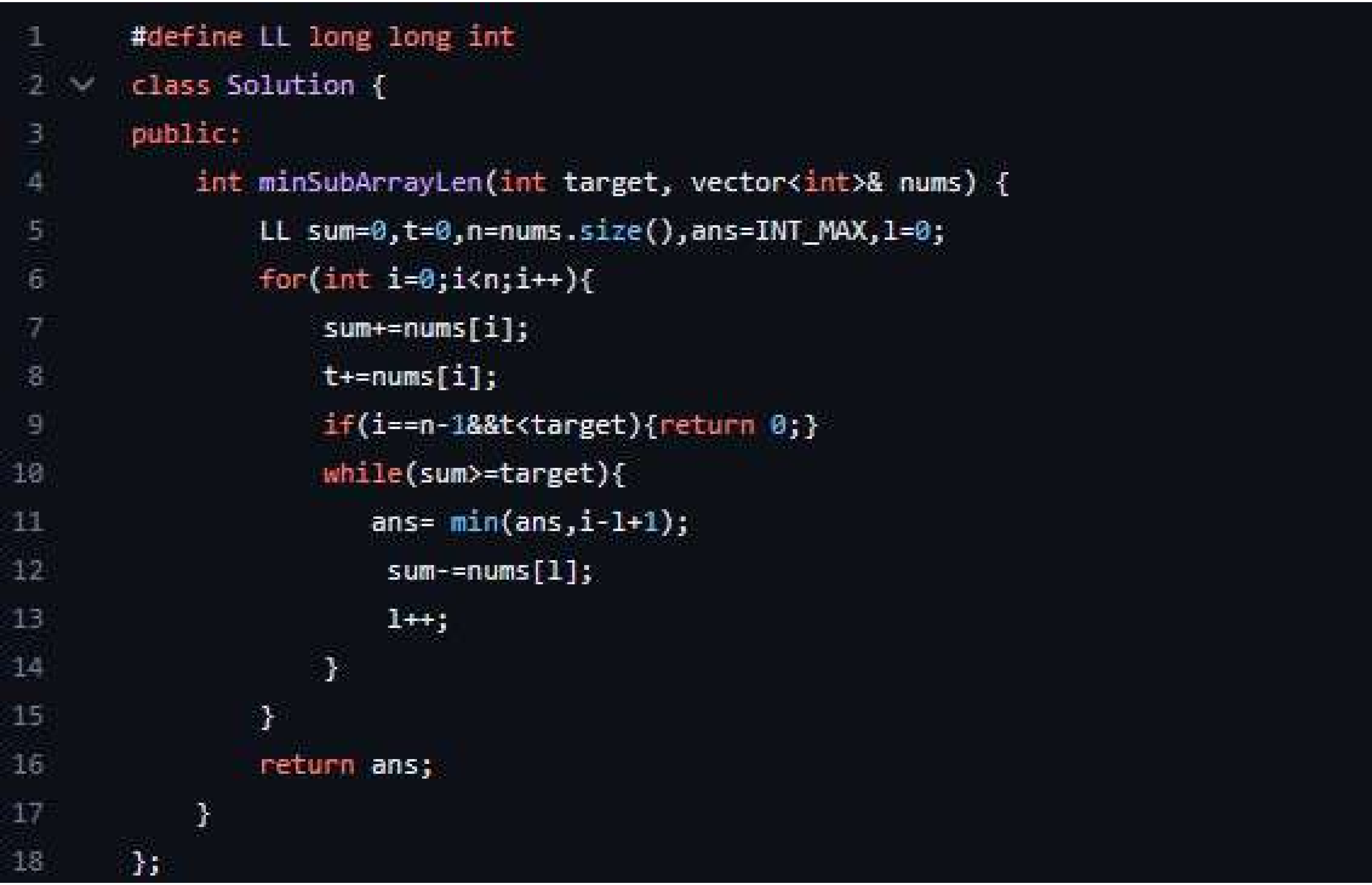
```
Input: target = 4, nums = [1,4,4]
Output: 1
```

Example 3:

```
Input: target = 11, nums = [1,1,1,1,1,1,1,1]
Output: 0
```

Constraints:

- `1 <= target <=` 10^9
- `1 <= nums.length <=` 10^5
- `1 <= nums[i] <=` 10^4



```
#define LL long long int
class Solution {
public:
    int minSubArrayLen(int target, vector<int>& nums) {
        LL sum=0,t=0,n=nums.size(),ans=INT_MAX,l=0;
        for(int i=0;i<n;i++){
            sum+=nums[i];
            t+=nums[i];
            if(i==n-1&&t<target){return 0;}
            while(sum>=target){
                ans= min(ans,i-l+1);
                 sum-=nums[l];
                 l++;
            }
        }
        return ans;
    }
};
```