930. Binary Subarrays With Sum

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 * 104`
- `nums[i]` is either `0` or `1`.
- `0 <= goal <= nums.length`

930. Binary Subarrays With Sum

Overview

We are given a binary array `nums` and an integer `goal`. The task is to find the number of non-empty subarrays in the given binary array where the sum of elements in the subarray equals the specified `goal`.

Key Observations:

1. The array contains only binary values (0 or 1).
2. The goal is to find subarrays with a specific sum.
3. The subarrays should be non-empty and contiguous.

Consider the given example with `nums = [1,0,1,0,1]` and `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**]

Note that all these subarrays are contiguous parts of the given array, and the count of such subarrays is the output.

930. Binary Subarrays With Sum

```
/*
   Prefix sum+two pointers (at most(goal)-at most(goal-1))
   Time complexity: O(n)
   Space complexity: O(1)
*/
class Solution {
public:
   int numSubarraysWithSum(vector<int>& nums, int goal) {
      int n=nums.size();

      auto at_most=[&](int k)->int{
         int ans=0;
         int l=0,prefix_sum=0;
         // Expand the window from the right
         for(int r=0;r<n;++r){
            prefix_sum+=nums[r];
            // Shrink the window while sum is greater than k
            while(l<=r && prefix_sum>k){
               prefix_sum-=nums[l]; // Update the sum
               l++; // Shrink the window from the left
            }

            // Compute the number of subarrays with sum at most k
            ans+=r-l+1;
         }

         return ans;
      };

      // Illustrated by Venn diagram
      return at_most(goal)-at_most(goal-1);
   }
};
```

930. Binary Subarrays With Sum

```
/*
   Prefix sum+two pointers (at least(goal)-at least(goal+1))
   Time complexity: O(n)
   Space complexity: O(1)
*/
class Solution {
public:
   int numSubarraysWithSum(vector<int>& nums, int goal) {
      int n=nums.size();

      auto at_least=[&](int k)->int{
         int ans=0;
         int l=0,prefix_sum=0;
         for(int r=0;r<n;++r){
            prefix_sum+=nums[r];
            while(l<=r && prefix_sum>=k){
               // Compute the number of subarrays with sum at least k
               ans+=n-r;
               prefix_sum-=nums[l];
               l++;
            }
         }
         return ans;
      };

      // Illustrated by Venn diagram
      return at_least(goal)-at_least(goal+1);
   }
};
```

3306. Count of Substrings Containing Every Vowel and K Consonants II

You are given a string `word` and a **non-negative** integer `k`.

Return the total number of of `word` that contain every vowel (`'a'`, `'e'`, `'i'`, `'o'`, and `'u'`) **at least** once and **exactly** `k` consonants.

Example 1:

Input: word = "aeioqq", k = 1

Output: 0

Explanation:

There is no substring with every vowel.

Example 2:

Input: word = "aeiou", k = 0

Output: 1

Explanation:

The only substring with every vowel and zero consonants is `word[0..4]`, which is `"aeiou"`.

Example 3:

Input: word = "ieaouqqieaouqq", k = 1

Output: 3

Explanation:

The substrings with every vowel and one consonant are:

- `word[0..5]`, which is `"ieaouq"`.
- `word[6..11]`, which is `"qieaou"`.
- `word[7..12]`, which is `"ieaouq"`.

Constraints:

- `5 <= word.length <= 2 * 105`
- `word` consists only of lowercase English letters.
- `0 <= k <= word.length - 5`

3306. Count of Substrings Containing Every Vowel and K Consonants II

```
/*
   Prefix sums+two pointers+at last(k)-at last(k+1)
   Time compelxity: O(n)
   Space compelxity: O(1)

*/

typedef long long ll;
typedef std::vector<int> vi;
typedef std::vector<vi> vvi;
class Solution {
   public:
      ll countOfSubstrings(std::string word, int k) {
         int n=word.size();

         auto is_vowel=[&](char letter)->bool{
            return letter=='a' || letter=='e' || letter=='i'|| letter=='o' || letter=='u';
         };

         // Funtion to check if we have, in range[l,r], at least one occurence of a vowel, and
         // at least k consonant.
         auto is_possible=[&](int& count_a,
                              int& count_e,
                              int& count_i,
                              int& count_o,
                              int& count_u,
                              int& count_cons,
                              int k)->bool{
            return count_a>=1 &&
               count_e>=1 &&
               count_i>=1 &&
               count_o>=1 &&
               count_u>=1 &&
               count_cons>=k;
         };
```

```
        // Function to check if we have, in the whole word, at least one occurrence of a vowel, and
        // at least k consonant.
        auto at_least=[&](int k)->ll{
            ll ans=0;
            int l=0;
            int count_a=0,count_e=0,count_i=0,count_o=0,count_u=0,count_cons=0;
            for(int r=0;r<n;++r){
                count_a+=(word[r]=='a');
                count_e+=(word[r]=='e');
                count_i+=(word[r]=='i');
                count_o+=(word[r]=='o');
                count_u+=(word[r]=='u');
                count_cons+=(!is_vowel(word[r]));
                while(is_possible(count_a,count_e,count_i,count_o,count_u,count_cons,k)){
                    count_a-=(word[l]=='a');
                    count_e-=(word[l]=='e');
                    count_i-=(word[l]=='i');
                    count_o-=(word[l]=='o');
                    count_u-=(word[l]=='u');
                    count_cons-=(!is_vowel(word[l]));
                    ans+=n-r;
                    l++;
                }
            }
            return ans;
        };


        return at_least(k)-at_least(k+1);
    }
};
```