Count Subarrays with given XOR

BRUTE FORCE: (TLE)

```
long subarrayXor(vector<int> &arr, int k) {
    int count=0;
    int n=arr.size();
    for(int i=0;i<n;i++){
      int x=0;
      for(int j=i;j<n;j++){
        x=x^arr[j];
        if(x==k){
          count++;
        }
      }
    }
    return count;
  }
```

Time Complexity: $O(n^2)$ — two nested loops
Space Complexity: O(1) — constant extra space used

OPTIMAL: (Using Hash Map and Prefix Sum)

```
int countSubarraysWithXOR(vector<int>& arr, int k) {
  unordered_map<int, int> freq; // Stores frequency of prefix XORs
  int count = 0;           // Total count of subarrays with XOR = k
  int xorSum = 0;          // Running prefix XOR

  for (int i = 0; i < arr.size(); i++) {
    xorSum ^= arr[i]; // Update the prefix XOR up to index i

    // If the prefix XOR itself equals k, we found a valid subarray from index 0 to i
    if (xorSum == k)
      count++;
```

```
    // Check if there exists a prefix XOR such that:
    // (prefixXOR till some j) ^ (prefixXOR till i) = k
    // => prefixXOR till j = xorSum ^ k
    if (freq.find(xorSum ^ k) != freq.end()) {
      count += freq[xorSum ^ k]; // Add the number of such occurrences to count
    }

    // Record the current prefix XOR in the map for future use
    freq[xorSum]++;
  }

  return count; // Return the total count of valid subarrays
}
```

Time Complexity: O(n)

Space Complexity: O(n) — for storing prefix XOR frequencies in the unordered map.

From the properties of XOR: $C = A \oplus B$
This implies: $A = C \oplus B$

arr = [4, 2, 2, 6, 4], k = 6

xorSum: prefix XOR up to current index

freq: map to store how many times each prefix XOR occurred

🧠 **Step-by-step Dry Run:**

i	arr[i]	xorSum ^= arr[i]	xorSum	xorSum == k?	xorSum^k	freq[xorSum^k]	count	freq map
0	4	0 ^ 4	4	❌	2	0	0	{4:1}
1	2	4 ^ 2	6	✅ ✅	0	0	1	{4:1, 6:1}
2	2	6 ^ 2	4	❌	2	1 (from i=1)	2	{4:2,

i	arr[i]	xorSum ^= arr[i]	xorSum	xorSum == k?	xorSum^k	freq[xorSum^k]	count	freq map
								6:1}
3	6	4 ^ 6	2	❌	4	2 (from i=0,2)	4	{4:2, 6:1, 2:1}
4	4	2 ^ 4	6	✅ ✅	0	0	5	{4:2, 6:2, 2:1}

GROUP ANAGARAMS:

```
vector<vector<string>> groupAnagrams(vector<string>& strs) {
  unordered_map<string, vector<string>> mp;  // Map to group anagrams by sorted string

  for (string s : strs) {
    string temp = s;                  // Copy original string
    sort(temp.begin(), temp.end());       // Sort characters to get the anagram key
    mp[temp].push_back(s);            // Group all anagrams with the same key
  }

  vector<vector<string>> ans;
  for (auto it : mp) {
    ans.push_back(it.second);          // Collect all anagram groups into result
  }

  return ans;                        // Return grouped anagrams
}
```

🧠 **Notes to Keep in Mind:**

1. ☑ **Sorting makes all anagrams have the same key:**
   - **E.g., "eat", "tea", "ate" → all become "aet" after sorting.**
2. ☑ **Unordered map groups anagrams efficiently:**
   - **Uses the sorted string as a hash key.**
   - **Automatically builds groups as you insert into the map.**
3. ⚠ **Sorting each string takes O(M log M):**
   - **Where M is the average length of the strings.**
   - **Total time is O(N * M log M) for N strings.**
4. ⚡ **Use this approach when:**
   - **String lengths are small or input size is manageable.**
5. ☑ **Output order doesn't matter (as per problem statement):**
   - **You can return the groups in any order.**

Time Complexity: O(N * M log M), Space Complexity: O(N * M)
where N = number of strings and M = average length of each string.

Encode and Decode Strings

```
// Function to encode a list of strings into a single string
string encode(vector<string>& s) {
  string encoded;

  for (string st : s) {
    // ':' is a delimiter so we can identify where the actual string starts
    encoded += to_string(st.length()) + ":" + st;
  }

  return encoded;
}
```

```
// Function to decode the encoded string back to list of strings
vector<string> decode(string& s) {
  vector<string> ans;
  int i = 0;

  while (i < s.length()) {
    int j = i;

    // Move j to find the ':' that separates length from string
    while (s[j] != ':') {
      j++;
    }

    // Extract the number between i and j (length of the next string)
    // substr(start, length) returns a substring
    int len = stoi(s.substr(i, j - i)); // stoi() converts substring to int

    // Extract the actual string of 'len' characters starting after ':'
    string temp = s.substr(j + 1, len);
    ans.push_back(temp);

    // Move i to the next encoded segment
    i = j + 1 + len;
  }

  return ans;
}
```

🧠 STL Functions Used:

Function	Description
to_string(int)	Converts an integer to a string
substr(i, len)	Returns a substring starting at i of length len
stoi(string)	Converts a numeric string to an integer

📌 Important Notes:

- Always use a length prefix when strings can contain any character (including :, #, etc.).
- This method is safe for all 256 ASCII characters.
- Works even if the string is empty, or contains digits or symbols.
- Avoid using characters as delimiters alone (like just #) — prefer length + delimiter.

⏱ Time and Space Complexity (One-liner):

Time: O(N), Space: O(N) — where N is the total length of all strings combined.

✅ Problem:

Count the number of subarrays where:

sum of subarray % k == length of subarray

i.e. sum(arr[i..j]) % k == (j - i + 1)

Given:

arr = [1, 4, 2, 3, 5]

n = 5

k = 100

BRUTE:

```
for (int i = 0; i < n; i++) {
  int sum = 0;

  for (int j = i; j < n; j++) {
    sum += arr[j];        // cumulative sum of subarray [i...j]
    int rem = sum % k;     // remainder of subarray sum mod k
    int len = j - i + 1;    // length of the subarray

    if (rem == len) {
      count++;          // condition satisfied
    }
  }
}
```

🧠 **Time Complexity:**

- **Outer loop: O(n)**
- **Inner loop: O(n)**
- **Total: $O(n^2)$**

🧠 **Space Complexity:**

- **O(1)**

EASIER VERSION: (Good Subarrays)

Count the number of subarrays where:
sum[i...j] == (j - i + 1)
(i.e., subarray sum equals its length)

🔍 **Let's understand it step-by-step:**

Let's define:

- **sum = prefix sum up to index j → sum = prefix[j]**
- **length = j - i + 1**

So the subarray [i...j] is good if:

prefix[j] - prefix[i - 1] == j - i + 1

→ prefix[j] - j == prefix[i - 1] - (i - 1)

Now define:

key = prefix sum - index

So if:

prefix[j] - j == prefix[i - 1] - (i - 1)

Then:

key[j] == key[i-1]

Which means: if the current key = sum - (j + 1) matches some previous key, then we found a valid subarray.

💡 **Why (j + 1)?**

In C++, indexing is 0-based, but:

- **Prefix sum is calculated from index 0 to j**
- **Length from 0 to j is j + 1**

🔁 **Example:**

arr = [1, 1, 1]

j = 1

sum = arr[0] + arr[1] = 2

length = j + 1 = 2

key = 2 - 2 = 0

CODE:

```
vector<int> arr = {1, 1, 1};

int n = 3;

int count = 0;

int sum = 0;


// 🔑 freq map tracks how many times a particular (sum - length) value has occurred

unordered_map<int, int> freq;

freq[0] = 1;  // Base case: empty prefix has sum 0 and length 0 → key = 0


for (int j = 0; j < n; j++) {

   sum += arr[j];        // Running prefix sum

   int len = j + 1;      // Subarray length from 0 to j

   int key = sum - len;   // 🔑 This key identifies valid subarrays: sum == length


   // ✅ If this key was seen before, we add all previous occurrences

   //    Each previous index i with same key means sum[i+1...j] == length[i+1...j]

   count += freq[key];


   // 🔁 Store/update frequency of current key

   freq[key]++;

}
```

OPTIMAL: (Good Subarrays modulo version)

```
vector<int> arr = {1, 2, 3, 4, 1};

int k = 4;

int count = 0;

int sum = 0;
```

```
// freq[key] stores how many times a particular mod pattern has occurred
unordered_map<int, int> freq;
freq[0] = 1;  // base case

for (int j = 0; j < arr.size(); j++) {
   sum += arr[j];            // running prefix sum
   int len = (j + 1) % k;       // length of subarray ending at j, mod k
   int key = ((sum % k) - len + k) % k;  // normalized key

   count += freq[key];         // all earlier matches are valid subarrays
   freq[key]++;              // record this key occurrence
}

cout << "Count = " << count << endl;
```

✅ **Fix: Always Normalize with +k before %k**

Update this line:

int key = ((sum % k) - len + k) % k;

✅ **This ensures key is always in the range [0, k-1].**

Time Complexity: O(n)

Space Complexity: O(k) — where n is the size of the array and k is the modulo base.

(a + b)%k = (a%k + b%k)%k

-> (a - b)%k = (a%k - b%k + k)%k;