Count number of nice subarrays

Problem Statement: Given an array nums and an integer k. An array is called nice if and only if it contains k odd numbers. Find the number of nice subarrays in the given array nums. A subarray is continuous part of the array.

Examples

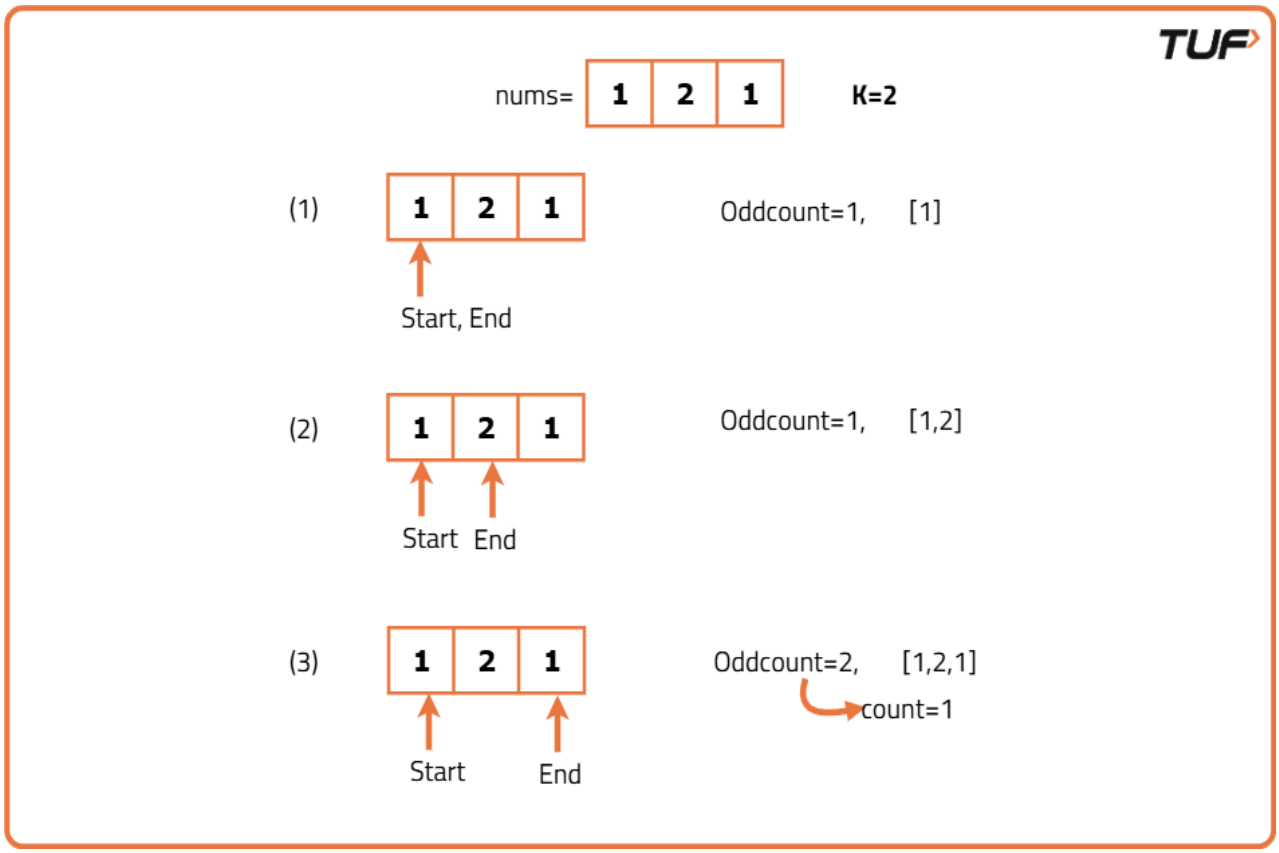
```
Input :nums = [1, 1, 2, 1, 1] , k = 3
Output :2
Explanation :The subarrays with three odd numbers are [1, 1, 2, 1] [1, 2, 1, 1]
Input : nums = [4, 8, 2] , k = 1
Output :0
Explanation :The array does not contain any odd number.
```

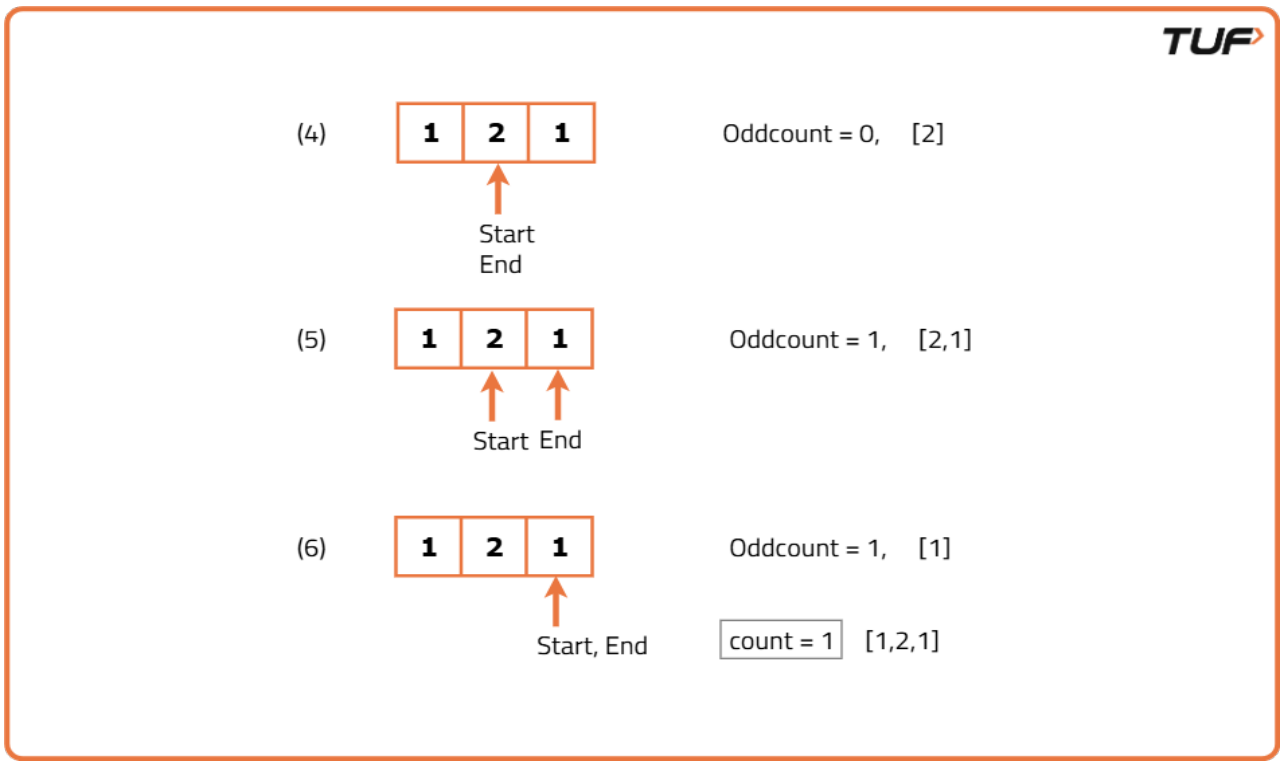
Brute force Approach

Algorithm

We are asked to count subarrays with exactly k odd numbers. The most basic way is to try out every possible subarray and count the odd numbers in it. If a subarray has exactly k odd numbers, we count it as a valid "nice" subarray. Think of this like manually scanning every possible window of the array and checking how many odd numbers are inside.

- Loop over all possible starting indices of the subarray.
- From each starting index, expand the subarray to the right, one element at a time.
- Keep track of how many odd numbers are in the current subarray.
- If the number of odd numbers equals k, increment the count of nice subarrays.
- If the number of odd numbers exceeds k, stop exploring this subarray further.
- Return the final count after trying all possibilities.





Code

```
import java.util.*;
class Solution {
    // Function to count subarrays with exactly k odd numbers
    public int numberOfSubarrays(int[] nums, int k) {
        // Initialize counter for total nice subarrays
        int count = 0;

        // Loop over all starting indices
        for (int start = 0; start < nums.length; start++) {
            // Track number of odd elements in current subarray
            int oddCount = 0;

            // Loop over ending indices starting from 'start'
            for (int end = start; end < nums.length; end++) {
                // Check if current number is odd
                if (nums[end] % 2 != 0)
                    oddCount++;

                // If odd count exceeds k, break (not nice)
                if (oddCount > k)
                    break;

                // If odd count is exactly k, count this subarray
                if (oddCount == k)
                    count++;
            }
        }

        // Return total nice subarrays
        return count;
    }
}

// Separate driver class with main method
public class Main {
    public static void main(String[] args) {
        Solution sol = new Solution();
```

```
        int[] nums = {1, 1, 2, 1, 1};
        int k = 3;
        System.out.println(sol.numberOfSubarrays(nums, k));
    }
}
```

Complexity Analysis

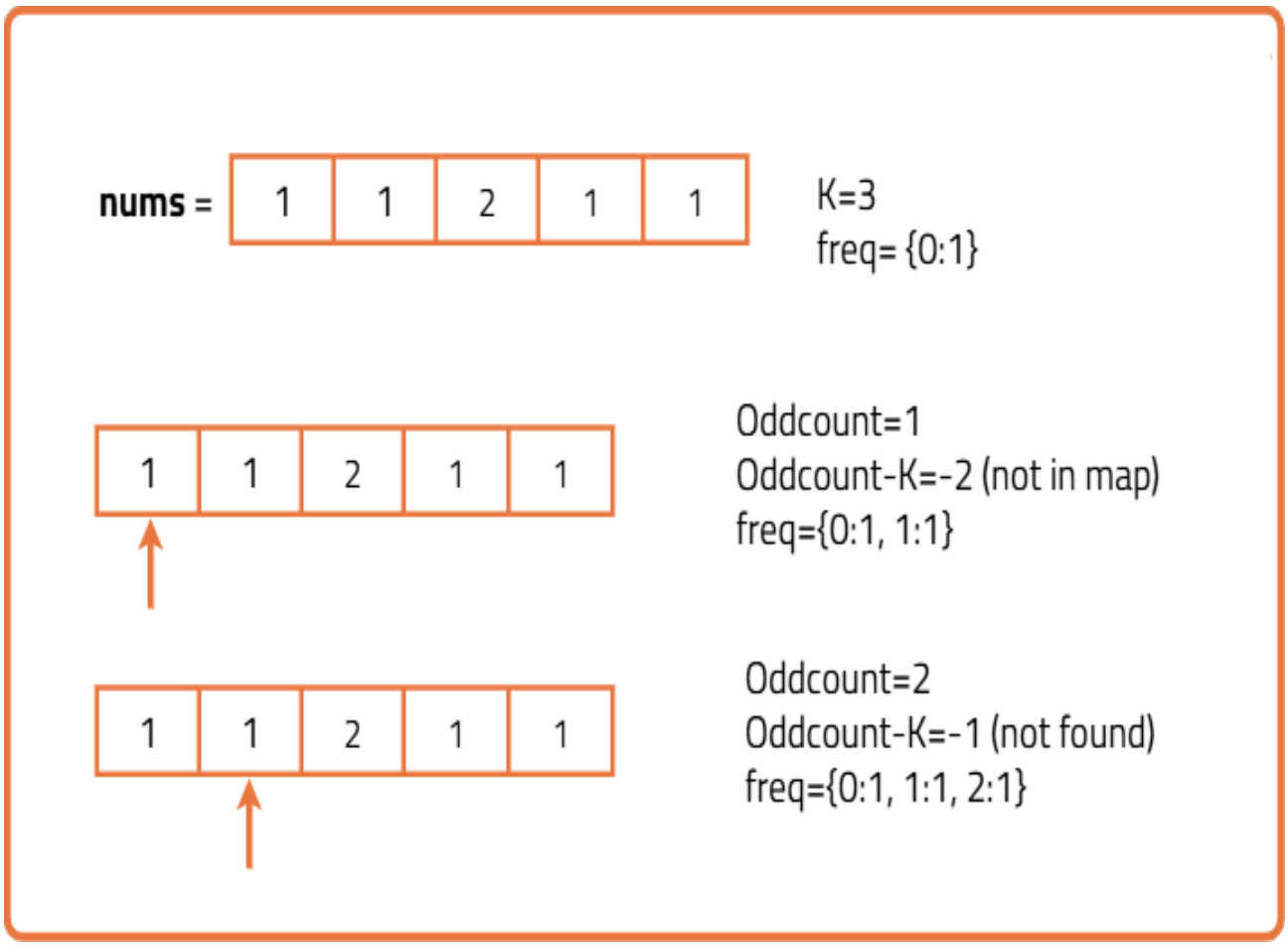
Time Complexity: O(N2) ,We use two nested loops to check all possible subarrays. For each subarray, we count the number of odd elements. The outer loop runs from index 0 to N-1, and the inner loop also runs up to N in the worst case. So total iterations can be approximately N * N = **O(N2)**.

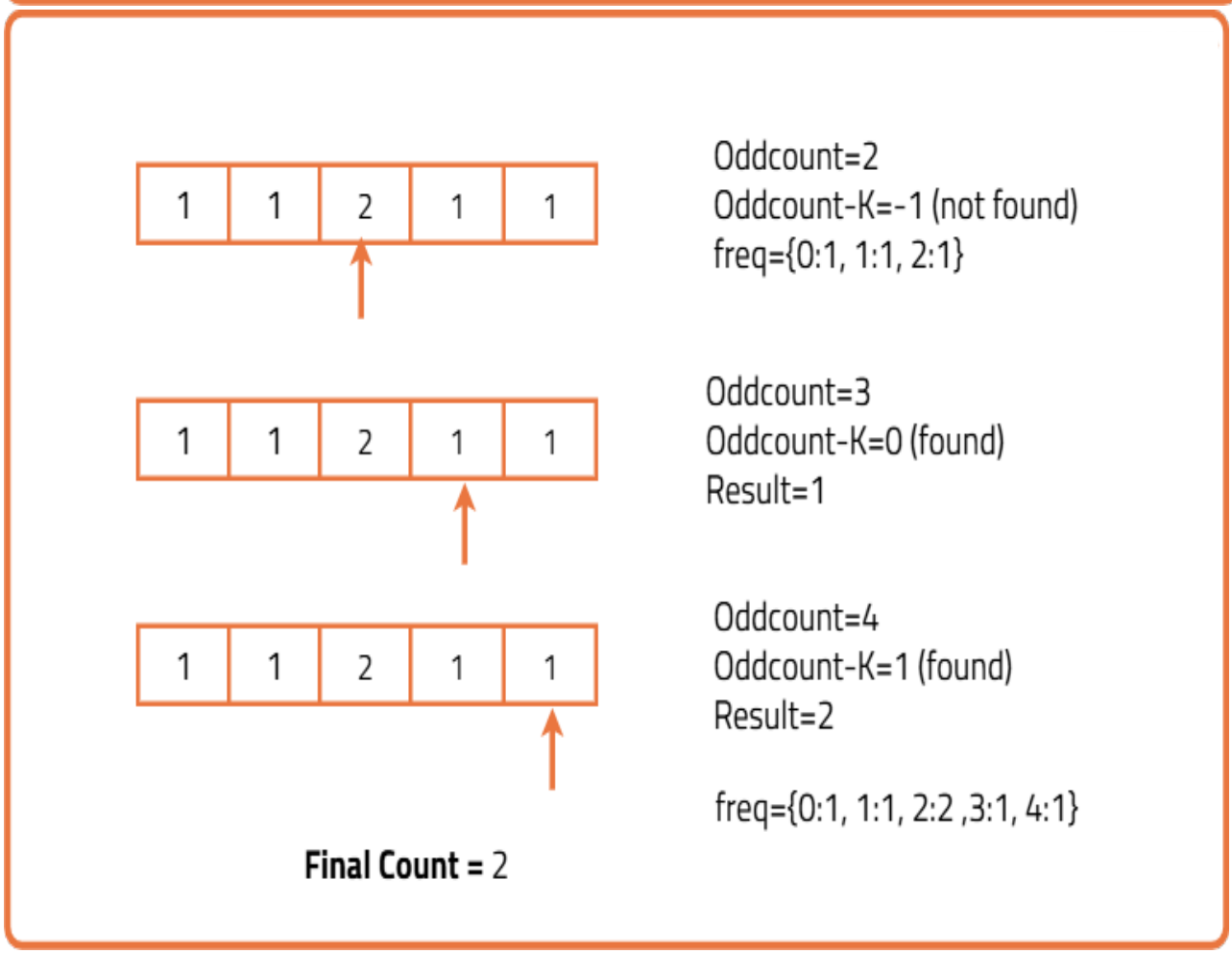
Space Complexity: O(1), No extra space used.

Better Approach
Algorithm
Instead of checking every possible subarray, we can speed things up by using prefix sums and a hashmap. We focus on counting how many subarrays have exactly `k` odd numbers. So, for each index we track how many odd numbers we've seen so far. If we've seen `oddCount` odds up to index `i`, and there was a previous prefix where we had `(oddCount-k)`, then the subarray between those two points has exactly `k` odd numbers.

We use a hashmap to store the count of how many times each `oddCount` has occurred. As we traverse the array, we just lookup how many times we've seen `(oddCount-k)` and add that to the answer.

- Use a frequency map to track how often a certain count of odd numbers has occurred in the prefix of the array.
- Start by adding a base case to the map: zero odd numbers seen so far has occurred once.
- Start traversing the array from left to right:
    - For each element, check if it is odd. If it is, increase the running count of odd numbers seen so far.
    - Check if the frequency map contains the value equal to the current count of odd numbers minus the required count. If yes, it means there are subarrays ending at the current index with exactly the required count of odd numbers. Add the frequency of that value to the final answer.
    - Update the frequency map by increasing the count of the current odd-number total.
- Once the entire array is processed, return the accumulated answer as the total number of valid subarrays.


nums = 1 1 2 1 1
K=3
freq= {0:1}
1 1 2 1 1
Oddcount=1
Oddcount-K=-2 (not in map)
freq={0:1, 1:1}
1 1 2 1 1
Oddcount=2
Oddcount-K=-1 (not found)
freq={0:1, 1:1, 2:1}


1 1 2 1 1
Oddcount=2
Oddcount-K=-1 (not found)
freq={0:1, 1:1, 2:1}
1 1 2 1 1
Oddcount=3
Oddcount-K=0 (found)
Result=1
1 1 2 1 1
Oddcount=4
Oddcount-K=1 (found)
Result=2
freq={0:1, 1:1, 2:2 ,3:1, 4:1}
Final Count = 2

Code

```
import java.util.*;

class Solution {
    // Function to count subarrays with exactly k odd numbers
    public int numberOfSubarrays(int[] nums, int k) {

        // Frequency map to track count of odd-number sums
        Map<Integer, Integer> freq = new HashMap<>();

        // Initial state: zero odd numbers has occurred once
        freq.put(0, 1);

        // Running count of odd numbers in current prefix
        int oddCount = 0;

        // Total number of nice subarrays
        int result = 0;

        // Traverse the entire array
        for (int num : nums) {

            // Check if number is odd
            if (num % 2 == 1) oddCount++;

            // Check if there's a prefix with (oddCount - k)
            if (freq.containsKey(oddCount - k)) {
                result += freq.get(oddCount - k);
            }

            // Update frequency map with current oddCount
            freq.put(oddCount, freq.getOrDefault(oddCount, 0) + 1);
        }

        // Return total result
        return result;
    }
}

// driver class
public class Main {
    public static void main(String[] args) {
        int[] nums = {1, 1, 2, 1, 1};
        int k = 3;

        Solution sol = new Solution();
        System.out.println(sol.numberOfSubarrays(nums, k));
    }
}
```

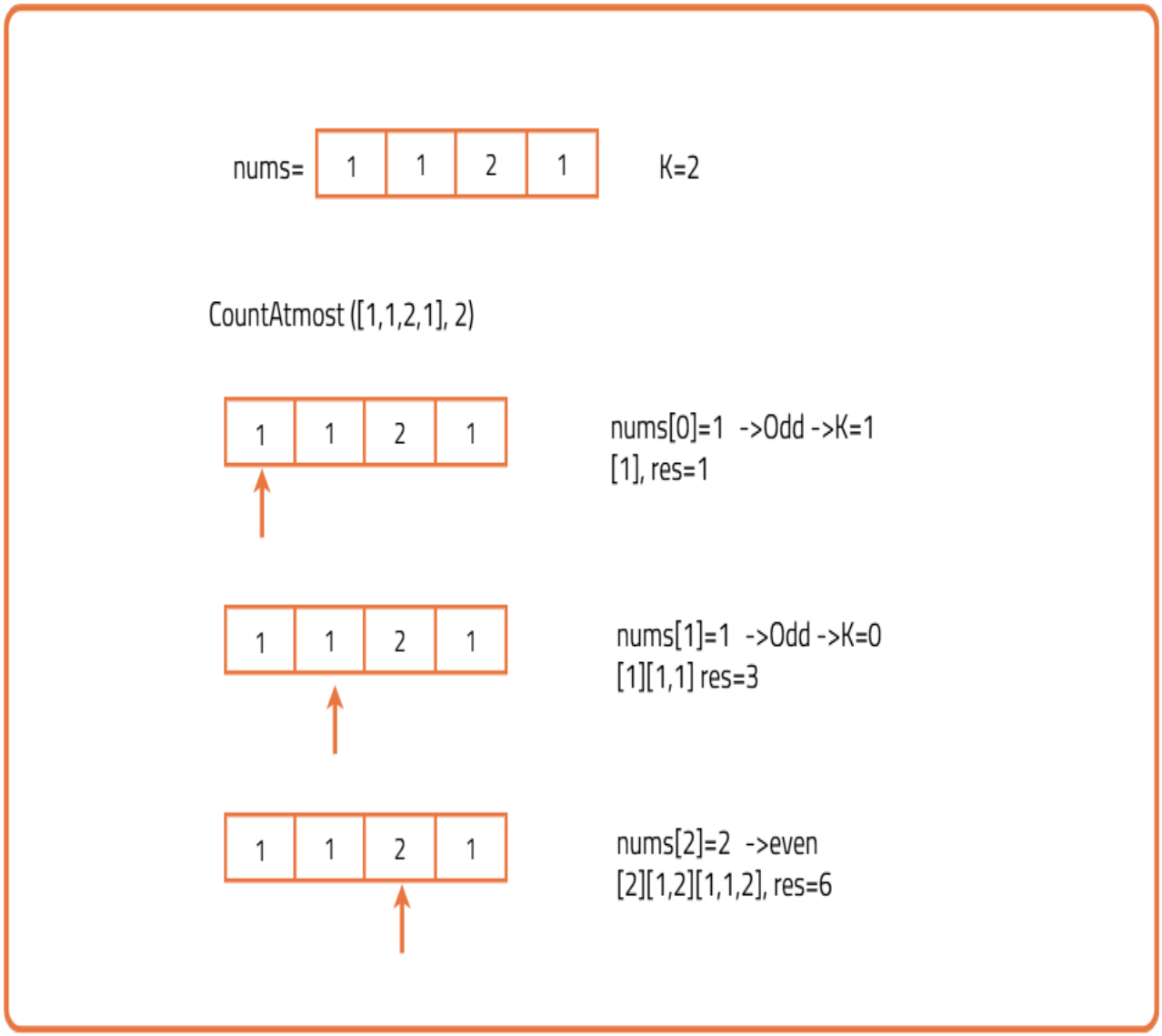
Complexity Analysis

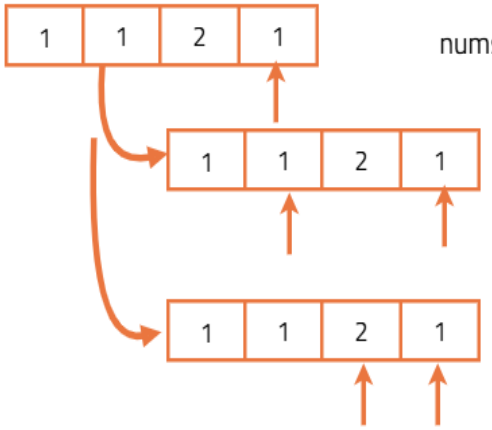
Time Complexity:O(N) ,We traverse the array once and each operation (map lookup, insertion, and update) takes constant time. So the total time complexity is linear in terms of the number of elements.

Space Complexity:O(N) ,We use a hashmap to store the frequency of prefix odd counts. In the worst case, all prefixes have different odd counts, leading to O(n) extra space.

Optimal Approach
Algorithm
We can't directly count subarrays with exactly K odd numbers using one pass of sliding window. But we can count how many subarrays have **at most** K odd numbers. If we do this for both K and K-1, then the difference gives us the number of subarrays that have **exactly K** odd numbers. This works because: `countExactlyK = countAtMost(K) - countAtMost(K - 1)`.

- Use a helper function to count the number of subarrays with at most a given number of odd numbers.
- Initialize two pointers to define the sliding window.
- Expand the window by moving the right pointer.
- If the number of odd numbers in the window exceeds the allowed count, move the left pointer to shrink the window.
- For each valid window, the number of subarrays ending at the current index is (right - left + 1).
- Repeat for K and K - 1, then return their difference.





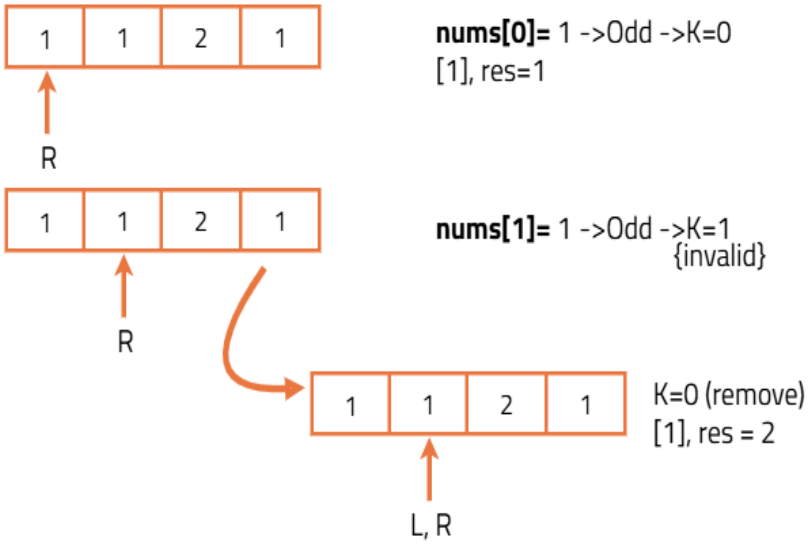
nums[3]=1 ->Odd ->K=-1 (Invalid)

K=0 (remove 1)

K=1 (remove 1)

[2][2,1], res=8

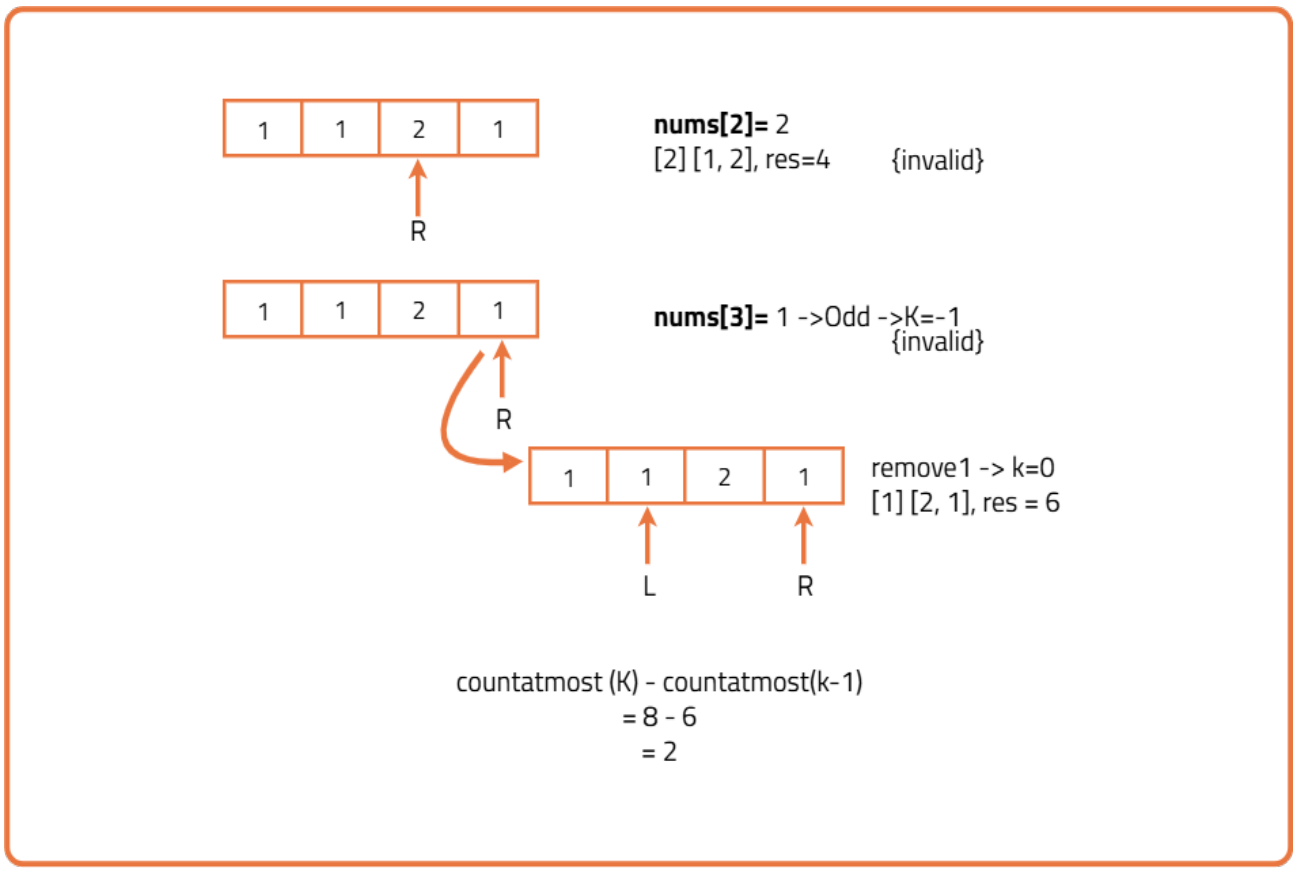
Count Atmost ([1, 1, 2, 1], 1)



nums[0]= 1 ->Odd ->K=0
[1], res=1

nums[1]= 1 ->Odd ->K=1
{invalid}

K=0 (remove)
[1], res = 2



Code

```
import java.util.*;

class Solution {
    // Helper function to count subarrays with at most k odd numbers
    public int countAtMost(int[] nums, int k) {
        int left = 0, res = 0;

        // Traverse through the array
        for (int right = 0; right < nums.length; right++) {
            // If current number is odd, reduce k
            if (nums[right] % 2 != 0)
                k--;

            // Shrink the window until k is valid
            while (k < 0) {
                if (nums[left] % 2 != 0)
                    k++;
                left++;
            }

            // Add valid subarrays ending at right
            res += (right - left + 1);
        }

        // Return result
        return res;
    }

    // Function to return number of subarrays with exactly k odd numbers
    public int numberOfSubarrays(int[] nums, int k) {
        return countAtMost(nums, k) - countAtMost(nums, k - 1);
    }
}
```

```
// Driver code in separate main class
public class Main {
    public static void main(String[] args) {
        Solution sol = new Solution();
        int[] nums = {1, 1, 2, 1, 1};
        int k = 3;
        System.out.println(sol.numberOfSubarrays(nums, k));
    }
}
```

Complexity Analysis

Time Complexity:O(n) ,We scan the array two times using the sliding window helper. Each scan processes every element at most once, making it linear in size of input.

Space Complexity:O(1) ,No additional space is used except a few integer variables for tracking window bounds and counts. So, constant space.