Max Consecutive Ones III

Problem Statement: 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.

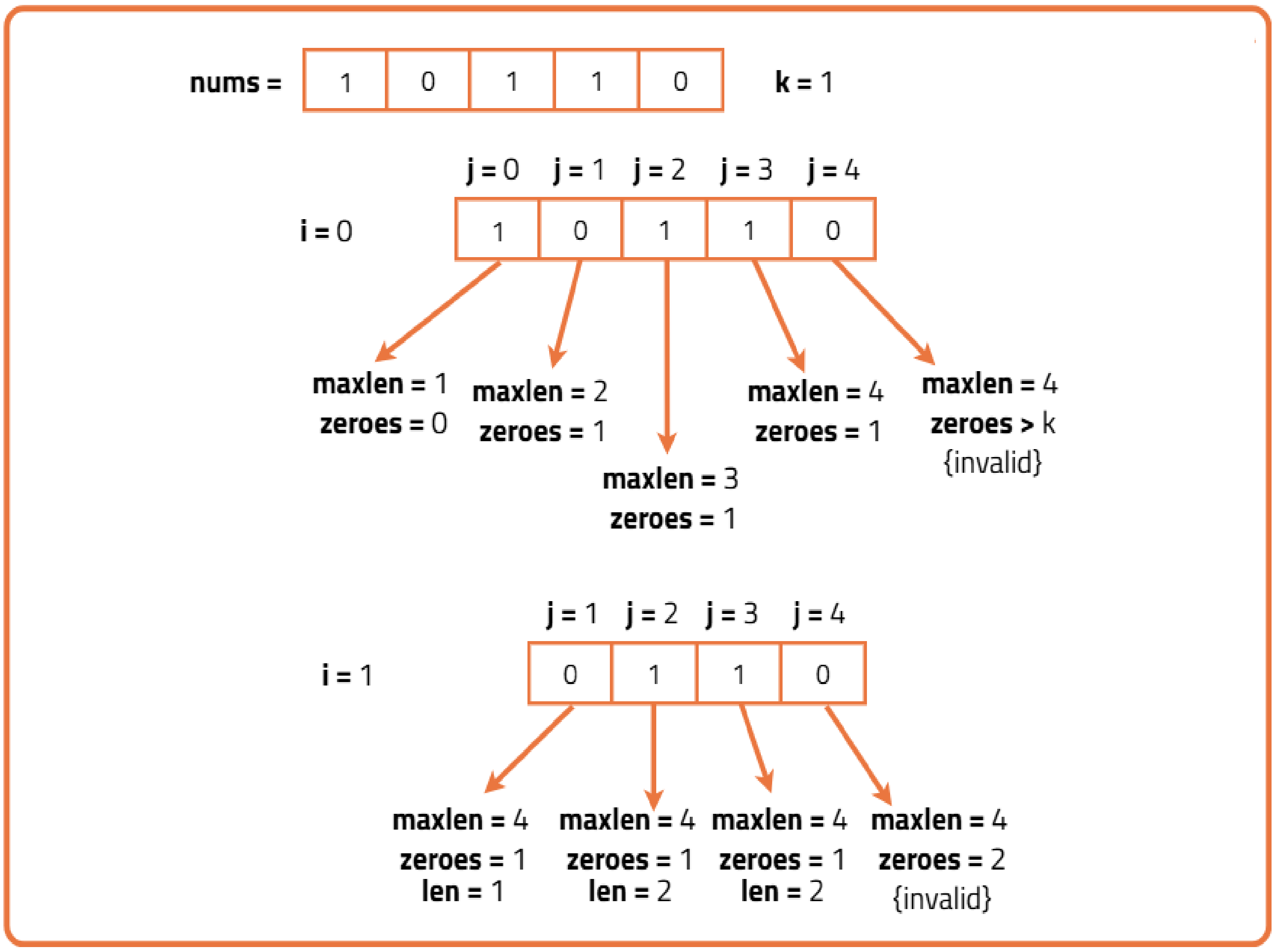
Examples

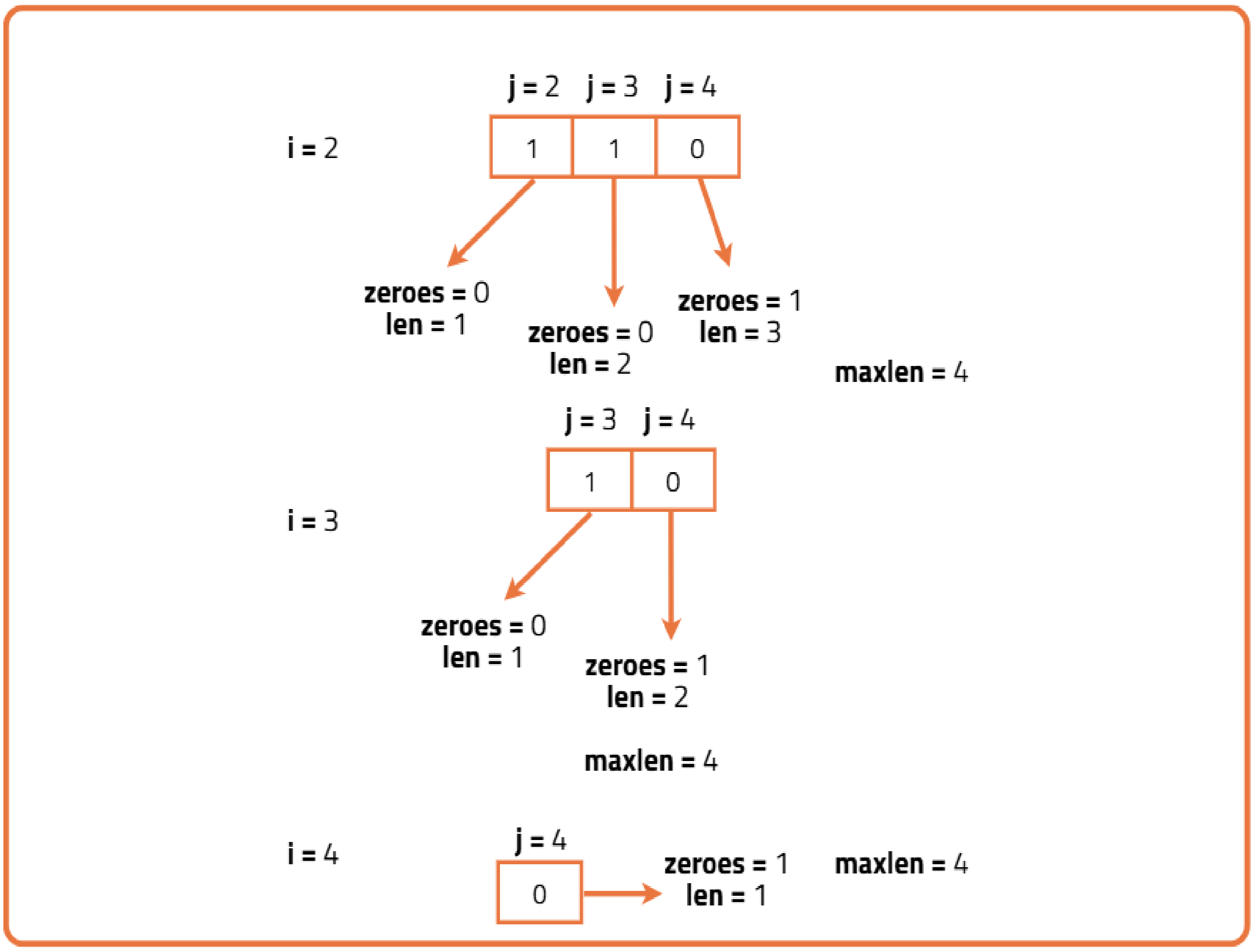
```
Input : nums = [1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0] , k = 3
Output : 10
Explanation : The maximum number of consecutive 1's are obtained only if we flip
the 0's present at position 3, 4, 5 (0 base indexing).
The array after flipping becomes [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0].
The number of consecutive 1's is 10.

Input : nums = [0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1] , k = 3
Output : 9
Explanation : The underlines 1's are obtained by flipping 0's in the new array.
[1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1].
The number of consecutive 1's is 9.
```

Brute Force Approach
Algorithm
In the brute force approach, we examine every possible subarray in the input array and check if the number of zeros in that subarray is less than or equal to the allowed number of flips (K). For each subarray that satisfies this condition, we calculate its length and update the maximum length encountered so far. Though inefficient, this approach checks all possibilities exhaustively and is useful for understanding the base logic of the problem.

- Iterate through all possible starting points of subarrays.
- For each starting point, iterate through all possible ending points to form subarrays.
- For every subarray, count the number of zeros present.
- If the number of zeros is less than or equal to `k`, treat the subarray as valid (since we can flip the zeros).
- Track the length of all valid subarrays and update the maximum length encountered.
- Return the length of the longest valid subarray found.


nums = 1 0 1 1 0
k = 1
j = 0 j = 1 j = 2 j = 3 j = 4
i = 0
1 0 1 1 0
maxlen = 1 zeroes = 0
maxlen = 2 zeroes = 1
maxlen = 3 zeroes = 1
maxlen = 4 zeroes = 1
maxlen = 4 zeroes > k {invalid}
j = 1 j = 2 j = 3 j = 4
i = 1
0 1 1 0
maxlen = 4 zeroes = 1 len = 1
maxlen = 4 zeroes = 1 len = 2
maxlen = 4 zeroes = 1 len = 2
maxlen = 4 zeroes = 2 {invalid}


j = 2 j = 3 j = 4
i = 2
1 1 0
zeroes = 0 len = 1
zeroes = 0 len = 2
zeroes = 1 len = 3
maxlen = 4
j = 3 j = 4
i = 3
1 0
zeroes = 0 len = 1
zeroes = 1 len = 2
maxlen = 4
i = 4
j = 4
0
zeroes = 1 len = 1
maxlen = 4

Code

```java
import java.util.*;
class Solution {
    // Function to find the longest subarray with at most k zero flips
    public int longestOnes(int[] nums, int k) {

        // Variable to store maximum length
        int maxLen = 0;

        // Loop to pick each possible start index
        for (int i = 0; i < nums.length; i++) {

            // Counter for zeros in current subarray
            int zeros = 0;

            // Loop to pick each end index for the subarray
            for (int j = i; j < nums.length; j++) {

                // If element is zero, increment zero counter
                if (nums[j] == 0) {
                    zeros++;
                }

                // If number of zeros exceeds allowed flips, break
                if (zeros > k) {
                    break;
                }

                // Update maximum length if current subarray is valid
                maxLen = Math.max(maxLen, j - i + 1);
            }
        }

        // Return the maximum valid subarray length
        return maxLen;
    }
}

// Driver code
public class Main {
    public static void main(String[] args) {
        Solution sol = new Solution();

        int[] nums = {1,1,1,0,0,0,1,1,1,1,0};
        int k = 2;

        // Output the result
        System.out.println(sol.longestOnes(nums, k));
    }
}
```

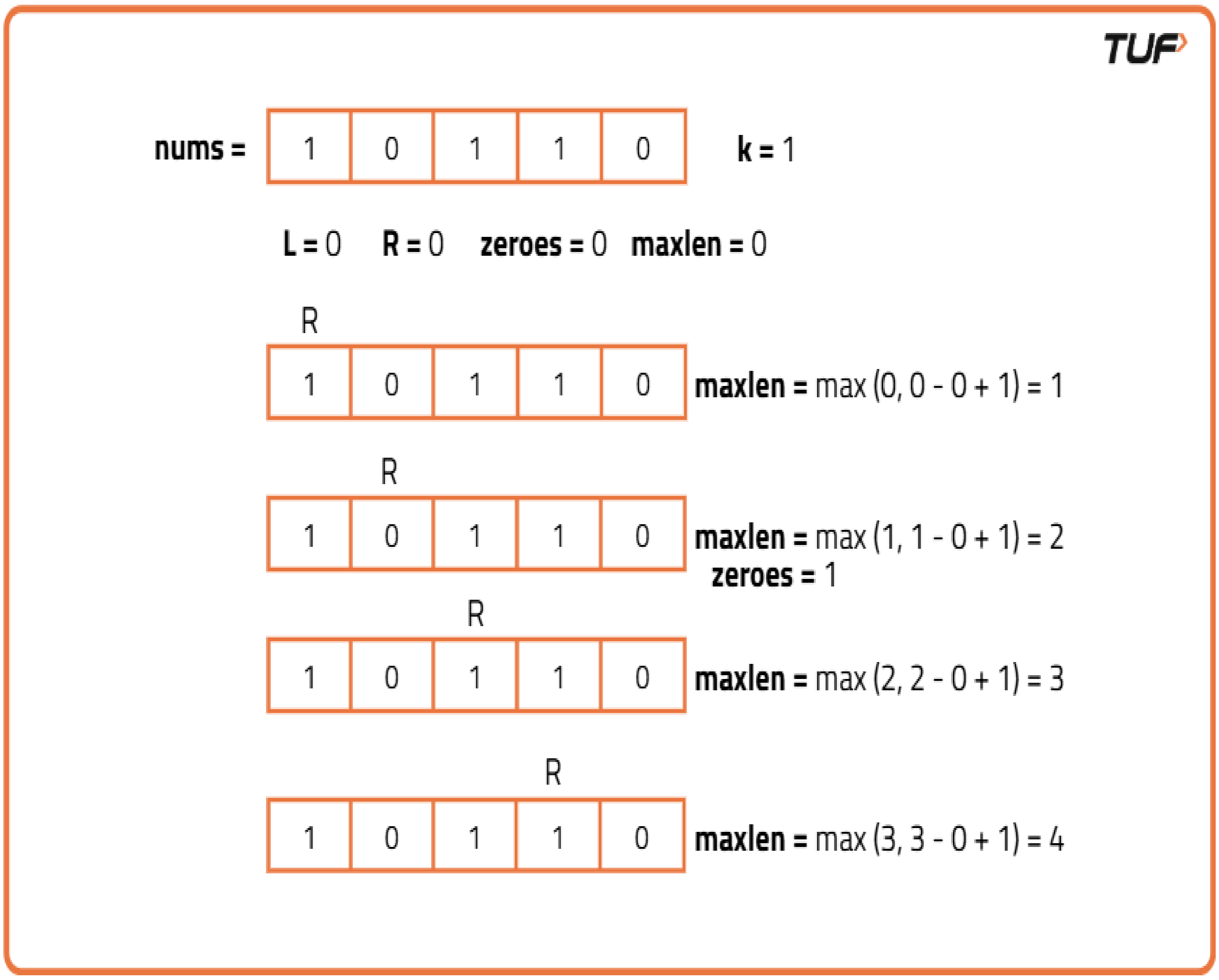
Complexity Analysis

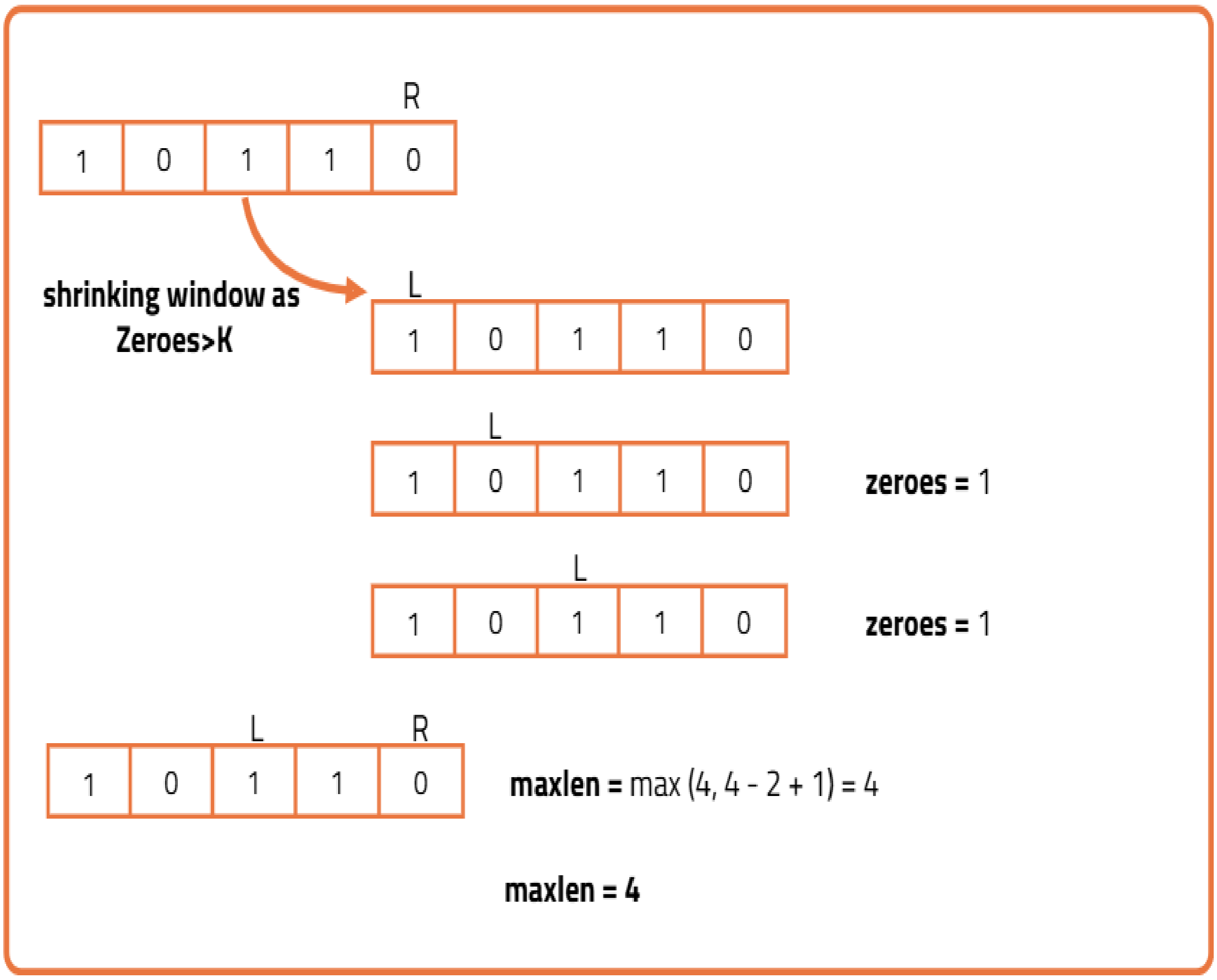
Time complexity: O(n^2) , as we are using two nested loops.

Space complexity: O(1), no extra space used.

Better Approach
Algorithm
To solve the problem efficiently, we can use a sliding window technique. The goal is to find the longest subarray of 1s that we can obtain by flipping at most K zeros. The idea is to expand the window to the right while keeping track of how many zeros are inside it. If the number of zeros becomes more than K, we shrink the window from the left until we are within the allowed limit of flips. At each step, we record the maximum window size that fits the requirement. This approach ensures we traverse the array only once.

- Initialize two pointers to represent the current window boundaries.
- Initialize a counter to keep track of the number of zeros in the window.
- Iterate through the array and expand the right boundary of the window.
- Each time a zero is encountered, increment the zero counter.
- If the zero counter exceeds the allowed limit, shrink the window from the left until the count is back within the limit.
- After each expansion, record the maximum window size encountered so far.


TUF
nums = 1 0 1 1 0 k = 1
L = 0 R = 0 zeroes = 0 maxlen = 0
R
maxlen = max (0, 0 - 0 + 1) = 1
R
maxlen = max (1, 1 - 0 + 1) = 2
zeroes = 1
R
maxlen = max (2, 2 - 0 + 1) = 3
R
maxlen = max (3, 3 - 0 + 1) = 4



Code

```java
import java.util.*;

class Solution {
    // Function to find the longest subarray with at most k zero flips
    public int longestOnes(int[] nums, int k) {

        // Left pointer for the window
        int left = 0;

        // Counter for zeros in the window
        int zeros = 0;

        // Variable to store maximum window length
        int maxLen = 0;

        // Right pointer expands the window
        for (int right = 0; right < nums.length; right++) {

            // If element is zero, increase the zero count
            if (nums[right] == 0) {
                zeros++;
            }

            // If zero count exceeds k, shrink the window
            while (zeros > k) {
                if (nums[left] == 0) {
                    zeros--;
                }
                // Move left pointer
                left++;
            }

            // Update the max length of the valid window
            maxLen = Math.max(maxLen, right - left + 1);
        }

        // Return the result
        return maxLen;
    }
}

// Driver code
public class Main {
    public static void main(String[] args) {
        Solution sol = new Solution();

        int[] nums = {1,1,1,0,0,0,1,1,1,1,0};
        int k = 2;

        // Output the result
        System.out.println(sol.longestOnes(nums, k));
    }
}
```

Complexity Analysis

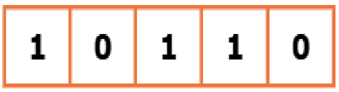
Time Complexity: O(N), We traverse the array only once using two pointers (left and right). Each element is visited at most twice once by the right pointer, and once by the left pointer when shrinking the window.

Space Complexity: O(1), Only a few integer variables are used to track the window and counters,so the space usage is constant.

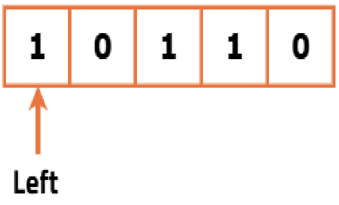
Optimal Approach
Algorithm
We can optimize the standard sliding window approach by eliminating the inner while-loop. Instead of using an explicit loop to shrink the window when the number of zeros exceeds the allowed flips (K), we can use a single conditional check to move the left pointer forward only when needed. This ensures that each pointer moves in a controlled and efficient manner without unnecessary loop nesting. The logic remains similar to the standard sliding window, but this structure can make the code slightly faster and cleaner in certain languages.

- Initialize two pointers, **left** and **right**, both set to 0, and a variable **zerocount** to keep track of the number of zeros in the current window.
- Traverse the array using the right pointer.
- If the current element is 0, increment **zerocount**.
- If **zerocount** exceeds **k**, check if the element at the left pointer is 0, and if so, decrement **zerocount**. Then increment the left pointer.
- At each step, calculate the current window size and update the maximum length if it's greater than the previously recorded maximum.
- Continue this process until the right pointer has traversed the entire array.
- Return the maximum window size found.

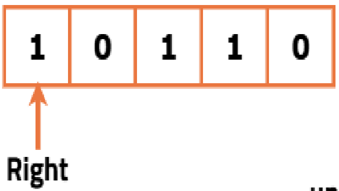
nums =



K = 1

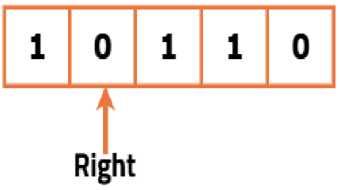


ZeroCount = 0 maxLen = 0
left = 0 right = 0



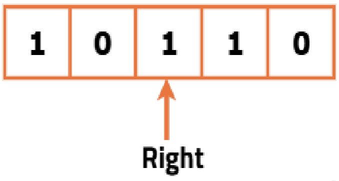
nums [0] = 1 zeroCount = 0 && <= k
(Valid Window)
Window = [0,0] --> length = 1

update, maxLen = 1



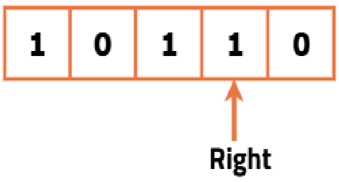
nums [1] = 0 zeroCount = 1 && <= k
(Valid Window)
Window = [0,1] --> length = 2

update, maxLen = 2



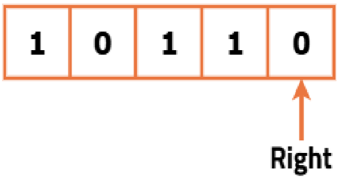
nums [2] = 1 not zero
Window = [0,2] --> length = 3

update, maxLen = 3



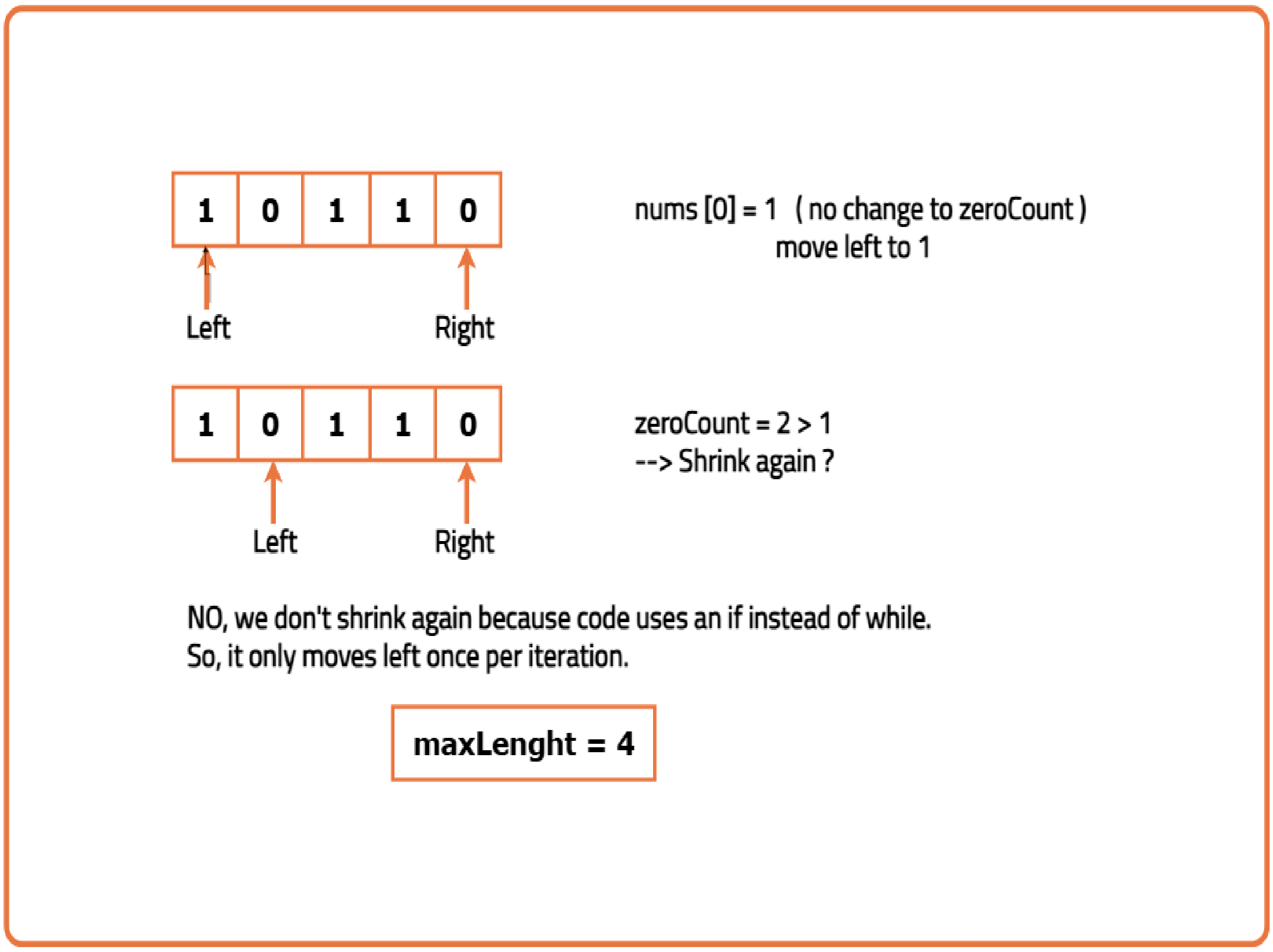
nums [3] = 1 not zero
Window = [0,3] --> length= 4

update, maxLen = 4



nums [4] = 0 zeroCount = 2 && > k
(Invalid Window)

Hence now we will shrink the Window from left


1 0 1 1 0
Left
Right
nums [0] = 1 (no change to zeroCount)
move left to 1
1 0 1 1 0
Left
Right
zeroCount = 2 > 1
--> Shrink again ?
NO, we don't shrink again because code uses an if instead of while.
So, it only moves left once per iteration.
maxLenght = 4

Code

```
import java.util.*;

class Solution {
    // Function to find the longest subarray with at most k zero flips
    public int longestOnes(int[] nums, int k) {

        // Left pointer of the sliding window
        int left = 0;

        // Counter for zeros in the window
        int zerocount = 0;

        // Variable to store maximum window length
        int maxlen = 0;

        // Right pointer expands the window
        for (int right = 0; right < nums.length; right++) {

            // If current element is zero, increment zerocount
            if (nums[right] == 0) {
                zerocount++;
            }

            // If zerocount exceeds k, move left and adjust zerocount
            if (zerocount > k) {
                if (nums[left] == 0) {
                    zerocount--;
                }
                // Shrink window from left
                left++;
            }
            // Update maximum window size
            maxlen = Math.max(maxlen, right - left + 1);
        }

        // Return the final result
        return maxlen;
    }
}

// Driver code
public class Main {
    public static void main(String[] args) {
        Solution sol = new Solution();

        int[] nums = {1,1,1,0,0,0,1,1,1,1,0};
        int k = 2;

        // Output the result
        System.out.println(sol.longestOnes(nums, k));
    }
}
```

Complexity Analysis

Time Complexity: O(N), Each element is processed at most once by the right pointer and once by the left pointer. There’s no nested iteration, so the traversal is strictly linear.

Space Complexity: O(1), We use only a fixed number of integer variables (left, right, zerocount, maxlen), regardless of input size, so space usage remains constant.