<https://leetcode.ca/2017-03-31-487-Max-Consecutive-Ones-II/>

Given a binary array nums, return the maximum number of consecutive 1's in the array if you can flip at most one 0.

**Example 1:**

**Input:** nums = [1,0,1,1,0]

**Output:** 4

**Explanation:**

- If we flip the first zero, nums becomes [1,1,1,1,0] and we have 4 consecutive ones.

- If we flip the second zero, nums becomes [1,0,1,1,1] and we have 3 consecutive ones.

The max number of consecutive ones is 4.

**Example 2:**

**Input:** nums = [1,0,1,1,0,1]

**Output:** 4

**Explanation:**

- If we flip the first zero, nums becomes [1,1,1,1,0,1] and we have 4 consecutive ones.

- If we flip the second zero, nums becomes [1,0,1,1,1,1] and we have 4 consecutive ones.

The max number of consecutive ones is 4.

**Constraints:**

1 <= nums.length <= 10^5

nums[i] is either 0 or 1.

**Follow up:** What if the input numbers come in one by one as an infinite stream? In other words, you can't store all numbers coming from the stream as it's too large to hold in memory. Could you solve it efficiently?

**Attempt 1: 2024-12-27**

**Solution 1: Not fixed length Sliding Window (10 min)**

public class Solution {

/\*\*

\* @param nums: a list of integer

\* @return: return a integer, denote the maximum number of consecutive 1s

\*/

public int findMaxConsecutiveOnes(int[] nums) {

int i = 0;

int zeroCount = 0;

int maxLen = 0;

for(int j = 0; j < nums.length; j++) {

if(nums[j] == 0) {

zeroCount++;

}

if(zeroCount > 1) {

if(nums[i] == 0) {

zeroCount--;

}

i++;

}

maxLen = Math.max(maxLen, j - i + 1);

}

return maxLen;

}

}

Time Complexity: O(n)

Space Complexity: O(1)

**Refer to chatGPT**

public class Solution {

public int findMaxConsecutiveOnes(int[] nums) {

int left = 0, maxLength = 0;

int zeroCount = 0;

for (int right = 0; right < nums.length; right++) {

// If the current number is 0, increase the zero count

if (nums[right] == 0) {

zeroCount++;

}

// If zeroCount exceeds 1, shrink the window from the left

while (zeroCount > 1) {

if (nums[left] == 0) {

zeroCount--;

}

left++;

}

// Update the maximum length

maxLength = Math.max(maxLength, right - left + 1);

}

return maxLength;

}

}

**Key Points in the Solution:**

Two Pointers:

Use left and right pointers to define the current sliding window.

Expand the window by incrementing right and shrink it by incrementing left.

Zero Count:

Track the number of 0s in the current window using zeroCount.

If zeroCount exceeds 1 (because only one 0 can be flipped), move left to reduce the window size until zeroCount <= 1.

Update Maximum:

Calculate the maximum length of the window whenever a valid window (with at most one 0) is maintained.

**What about follow up ?**

Follow up: What if the input numbers come in one by one as an infinite stream? In other words, you can't store all numbers coming from the stream as it's too large to hold in memory. Could you solve it efficiently?

**Key Idea:**

Sliding Window for Streams:

**Maintain a sliding window using two variables:**

**one to track the length of the current sequence of 1s before the last 0 (countBeforeZero)**

**another to track the length of the current sequence after the last 0 (countAfterZero).**

**When a 0 is encountered, update the counts and keep the "window" moving forward.**

Tracking the Result:

At any point in the stream, the maximum consecutive 1s (with at most one flipped 0) is the sum of countBeforeZero + countAfterZero + 1 (if a 0 is flipped).

Space Efficiency:

Instead of storing the entire stream, only keep two counters and a running maximum, making the solution highly space-efficient.

**Java Solution for Infinite Stream:**

public class Solution {

public int findMaxConsecutiveOnesStream(int[] stream) {

int countBeforeZero = 0; // Count of 1's before the last 0

int countAfterZero = 0; // Count of 1's after the last 0

int maxLength = 0; // Maximum length of consecutive 1's

boolean seenZero = false; // Whether we've seen a 0 in the current window

for (int num : stream) {

if (num == 1) {

countAfterZero++; // Extend the sequence after the last 0

} else {

// Shift the window: the current 'countAfterZero' becomes 'countBeforeZero'

countBeforeZero = countAfterZero;

countAfterZero = 0; // Reset the count for the new sequence

seenZero = true;

}

// Calculate the maximum length

if (seenZero) {

maxLength = Math.max(maxLength, countBeforeZero + countAfterZero + 1); // Flip one 0

} else {

maxLength = Math.max(maxLength, countAfterZero); // No 0 to flip

}

}

return maxLength;

}

}