# Question

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

**Example 1:**

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

**Output:** 4

**Explanation:** Flip the first zero will get the the maximum number of consecutive 1s.

After flipping, the maximum number of consecutive 1s is 4.

**Note:**

* The input array will only contain 0 and 1.
* The length of input array is a positive integer and will not exceed 10,000

**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?

# Solution

**Intuition**

First, let's understand our problem.

"Given a binary array, find the maximum number of consecutive 1s in this array..."

Okay makes sense so far.

"...if you can flip at most one 0."

Huh? What does that even mean?

Let's translate that into something more concrete. We can rephrase "if you can flip at most one 0" into "allowing at most one 0 within an otherwise consecutive run of 1s". These statements are equal because if we had one 0 in our consecutive array, we could flip it to satisfy our condition. Note, we're not actually going to flip the 0 which will make our approach simpler.

So our new problem statement is:

"Given a binary array, find the maximum number of consecutive 1s in this array, allowing at most one 0 within an otherwise consecutive run of 1s"

#### **Approach 1: Brute Force**

**Algorithm**

Let's start simple and work our way up.

A brute force solution usually involves trying to check every single possibility. It'll look something like this:

* Check every possible consecutive sequence
* Count how many 0's are in each sequence
* If our sequence has one or fewer 0's, check if that's the longest consecutive sequence of 1's.

**Interview Tip:** Often times the interviewer doesn't need to see you code the brute force solution. State the brute force approach out loud and discuss his/her expectations. Either way, communicating proactively will give you major bonus points.

|  |
| --- |
| **class Solution {**  **public int findMaxConsecutiveOnes(int[] nums) {**  **int longestSequence = 0;**  **for (int left = 0; left < nums.length; left++) {**  **int numZeroes = 0;**  **// check every consecutive sequence**  **for (int right = left; right < nums.length; right++) {**  **// count how many 0's**  **if (nums[right] == 0) {**  **numZeroes += 1;**  **}**  **// # update answer if it's valid**  **if (numZeroes <= 1) {**  **longestSequence = Math.max(longestSequence, right - left + 1);**  **}**  **}**  **}**  **return longestSequence;**  **}**  **}** |

**Complexity Analysis**

Let n*n* be equal to the length of the input nums array.

* Time complexity : O(n^2)*O*(*n*2). The nested for loops turn our approach into a quadratic solution because for every index, we have to check every other index in the array.
* Space complexity : O(1)*O*(1). We are using 4 variables: left, right, numZeroes, longestSequence. The number of variables are constant and do not change based on the size of the input.

#### **Approach 2: Sliding Window**

**Intuition**

The naive approach works but our interviewer is not convinced. Let's see how we can optimize the code we just wrote.

The brute force solution had a time complexity of O(n^2)*O*(*n*2). What was the bottleneck? Checking every single consecutive sequence. Intuitively, we know we're doing repeated work because sequences overlap. We are checking consecutive sequence blindly. We need to establish some rules on how to move our sequence forward.

* If our sequence is valid, let's continue expanding our sequence (because our goal is to get the largest sequence possible).
* If our sequence is invalid, let's stop expanding and contract our sequence (because an invalid sequence will never count towards our largest sequence).

The pattern that comes to mind for expanding/contracting sequences is the sliding window. Let's define valid and invalid states.

* Valid State = one or fewer 0's in our current sequence
* Invalid State = two 0's in our current sequence

**Algorithm**

Great. How do we apply all this to the sliding window?

Let's use left and right pointers to keep track of the current sequence a.k.a. our window. Let's expand our window by moving the right pointer forward until we reach a point where we have more than one 0 in our window. When we reach this invalid state, let's contract our window by moving the left pointer forward until we have a valid window again. By expanding and contracting our window from valid and invalid states, we are able to traverse the array efficiently without repeated overlapping work.

Now we can break this approach down into a few actionable steps:

While our window is in bounds of the array...

1. Add the rightmost element to our window
2. Check if our window is invalid. If so, contract the window until valid.
3. Update our the longest sequence we've seen so far
4. Continue to expand our window

This will look like this:

1 / 10

|  |
| --- |
| **class Solution {**  **public int findMaxConsecutiveOnes(int[] nums) {**  **int longestSequence = 0;**  **int left = 0;**  **int right = 0;**  **int numZeroes = 0;**  **// while our window is in bounds**  **while (right < nums.length) {**  **// add the right most element into our window**  **if (nums[right] == 0) {**  **numZeroes++;**  **}**  **// if our window is invalid, contract our window**  **while (numZeroes == 2) {**  **if (nums[left] == 0) {**  **numZeroes--;**  **}**  **left++;**  **}**  **// update our longest sequence answer**  **longestSequence = Math.max(longestSequence, right - left + 1);**  **// expand our window**  **right++;**  **}**  **return longestSequence;**  **}**  **}** |

**Complexity Analysis**

Let n*n* be equal to the length of the input nums array.

* Time complexity : O(n)*O*(*n*). Since both the pointers only move forward, each of the left and right pointer traverse a maximum of n steps. Therefore, the timecomplexity is O(n)*O*(*n*).
* Space complexity : O(1)*O*(1). Same as the previous approach. We don't store anything other than variables. Thus, the space we use is constant because it is not correlated to the length of the input array.