1493. Longest Subarray of 1's After Deleting One Element <u>Dynamic Programming</u> <u>Sliding Window</u> Medium <u>Array</u>

The problem presents a binary array nums, comprised of 0s and 1s. The task is to find the size of the largest contiguous

Problem Description

such subarray made up entirely of 1s after removing an element, the function should return 0. Intuition

subarray of 1s that could be obtained by removing exactly one element from the array. You are also informed that if there is no

The intuition for solving this problem involves looking at each segment of 1s as a potential candidate and checking the length we

can obtain after removing an element. We want to maximize this length by strategically picking an element for removal. The key

insight here is that if there are consecutive 1s, removing one 1 does not impact the length of the subarray of 1s on either side of it. We can approach the solution by precomputing and storing the lengths of continuous 1s at any index from the left and right directions separately. This is done in two linear scans using the left and right auxiliary arrays: From Left to Right: Starting at the index 1, if the previous element is 1, increment the value in the left array at the current

index by 1 plus the value at the previous index. This will give us the number of continuous 1s to the left of each index in the array nums. From Right to Left: Starting from the second to last index and going backwards, if the next element is 1, increment the value

in the right array at the current index by 1 plus the value at the next index. This gives the number of continuous 1s to the right of each index. The final result, after having populated both left and right arrays, is the maximum sum of corresponding elements from left

and right arrays across all indices. This effectively simulates the removal of one element (either a 1 or 0) and joining the

- contiguous 1s from both left and right sides. However, the result should be the length after the removal of one element, so another -1 is implicitly considered in the max operation [since we don't actually remove the element but calculate what the length would be as if we did].
- By applying this method, we can find the longest subarray containing only 1s after deleting exactly one element. Solution Approach

The solution to this problem relies primarily on dynamic programming to keep track of the length of continuous 1s on both sides

Initialization: Two lists, left and right, each of the same length as nums, are created and initialized with zeroes. These lists

Populating the left list: We start iterating over the array nums from the second element onwards (index 1 because we

consider 0 as the base case which is already initialized to zero). For each index i, if the element at the previous index (i -

1) is 1, we set left[i] to left[i - 1] + 1. Effectively, this records the length of a stretch of 1s ending at the current

(index n - 2 where n is the length of nums). For each index i, if the element at the next index (i + 1) is 1, we set

the element at the current index. However, since we are supposed to actually remove an element to create the longest

subsequence, the sum implicitly considers this by not adding an additional 1 even though we are summing the lengths from

will store the length of continuous 1s to the left and right of each index, respectively.

Populating the right list: We do a similar iteration, but this time from right to left, starting from the second-to-last element

both sides.

Example Walkthrough

index.

of each index in the array nums.

Here is a breakdown of the implementation details:

right[i] to right[i + 1] + 1. This captures the length of a stretch of 1s starting right after the current index. Finding the longest subarray: Once we have both left and right lists populated, we iterate over all possible remove

positions (these positions do not have to be 1; they can be 0 as well, as we will merge stretches of 1s around them). For each index, we calculate the sum of left[i] + right[i], which approximates the length of the subarray of 1s if we removed

Let's use an example to illustrate the solution approach.

left: [0, 0, 0, 0, 0, 0, 0, 0]

right: [0, 0, 0, 0, 0, 0, 0, 0]

Populating the **left** list:

Populating the right list:

Finding the longest subarray:

Suppose the input binary array nums is [1, 1, 0, 1, 1, 1, 0, 1, 1].

5. Returning the result: The maximum value from these sums corresponds to the size of the longest subarray containing only 1s after one deletion. This is found using the max() function applied to the sums of left[i] + right[i] for each index i.

The algorithm completes this in linear time with two passes through nums, and the space complexity is governed by the two

additional arrays left and right, which are linear in the size of the input array nums.

Step by Step Solution Initialization: We initialize two lists, left and right, with the same length as nums.

We iterate from left to right over the array nums, starting from index 1. o nums[1] is 1, and nums[0] is 1, so left[1] = left[0] + 1 -> 0 + 1 = 1. As we continue, left becomes [0, 1, 0, 1, 2, 3, 0, 1, 2].

In this case, the maximum is 5, corresponding to index 3. Thus, the size of the longest subarray containing only 1s after one

This approach allows us to solve the problem with a single pass for left and another for right, totaling linear time complexity,

O(n), with n being the number of elements in nums. The use of the two lists left and right gives us a space complexity of O(n)

We calculate the potential longest subarrays after a deletion for each index, by summing the left and right values at each

deletion is 5.

Solution Implementation

n = len(nums)

left ones count = [0] * n

for i in range(1, n):

right_ones_count = [0] * n

if nums[i - 1] == 1:

for i in range(n - 2, -1, -1):

public int longestSubarray(int[] nums) {

int[] leftOnesCount = new int[length];

for (int i = 1; i < length; ++i) {</pre>

if (nums[i - 1] == 1) {

if (nums[i + 1] == 1) {

int maxSubarrayLength = 0;

const size: number = nums.length;

for (let i = 1; i < size; i++) {

if (nums[i - 1] === 1) {

if (nums[i + 1] === 1) {

for (let i = 0; i < size; i++) {

let maxLength: number = 0;

let left: number[] = new Array(size).fill(0);

left[i] = left[i - 1] + 1;

right[i] = right[i + 1] + 1;

for (let i = size - 2; i >= 0; i--) {

let right: number[] = new Array(size).fill(0);

// Create two arrays to keep track of consecutive 1's to the left and right

// Fill the left array with the counts of consecutive 1's from the left

// Fill the right array with the counts of consecutive 1's from the right

// Initialize the variable to store the maximum length of the subarray

// Iterate over the input array to compute the maximum subarray length

maxLength = Math.max(maxLength, left[i] + right[i]);

def longestSubarray(self, nums: List[int]) -> int:

Determine the length of the nums list

// The longest subarray is the sum of consecutive 1's to the left and right of the current element

int[] rightOnesCount = new int[length];

for (int $i = length - 2; i >= 0; --i) {$

int length = nums.length;

if nums[i + 1] == 1:

as well.

Python

index.

 \circ At index 6 (the second 0), the sum is left[6] + right[6] = 3 + 0 = 3. The longest subarray occurs at either index 2 or 6, where we would be removing the 0 to join the 1 s.

Determine the length of the nums list

Iterating from right to left, starting from index 7.

Returning the result: The maximum value from the sum of left and right for all indexes gives us the result.

Considering every index, we would have sums as [2, 1, 4, 5, 3, 2, 4, 1, 2].

• At index 2 (the first 0), the sum is left[2] + right[2] = 0 + 0 = 0.

o nums[7] is 1, and nums[8] is 1, so right[7] = right[8] + 1 -> 0 + 1 = 1.

Continuing this process, the right list fills up as [2, 1, 0, 4, 3, 2, 0, 1, 0].

from typing import List class Solution: def longestSubarray(self, nums: List[int]) -> int:

Calculate the consecutive ones to the left of each index

Calculate the consecutive ones to the right of each index

left_ones_count[i] = left_ones_count[i - 1] + 1

right_ones_count[i] = right_ones_count[i + 1] + 1

Find the maximum length subarray formed by summing up counts of left and right ones. # Note that the question assumes we can remove one zero to maximize the length. # So, connecting two streaks of ones effectively means removing one zero between them. max_length = max(a + b for a, b in zip(left_ones_count, right_ones_count)) return max_length Java

// Function to find the length of the longest subarray consisting of 1s after deleting exactly one element.

// Arravs to store the count of consecutive 1s to the left and right of each index in nums

// Count consecutive 1s from left to right, starting from the second element

// Count consecutive 1s from right to left, starting from the second-to-last element

maxSubarrayLength = Math.max(maxSubarrayLength, leftOnesCount[i] + rightOnesCount[i]);

// If the previous element is 1, increment the count

leftOnesCount[i] = leftOnesCount[i - 1] + 1;

// If the next element is 1, increment the count

// Variable to store the answer, the maximum length of a subarray

rightOnesCount[i] = rightOnesCount[i + 1] + 1;

Initialize two lists to keep track of consecutive ones to the left and right of each index

// Loop to find the maximum length by combining the left and right counts of 1s for (int i = 0; i < length; ++i) {</pre> // Compute the length of subarray by removing the current element, hence adding left and right counts of 1s. // Since one element is always removed, the combined length of consecutive 1s from left and right

class Solution {

```
// Reduce the length by 1 if the length of consecutive 1s equals the array length, since we need to remove one element.
        if (maxSubarrayLength == length) {
            maxSubarrayLength--;
        // Return the maximum length of subarray after deletion
        return maxSubarrayLength;
C++
class Solution {
public:
    int longestSubarray(vector<int>& nums) {
        // Get the size of the input array
        int size = nums.size();
        // Create two vectors to keep track of consecutive 1's on the left and right
        vector<int> left(size, 0):
        vector<int> right(size, 0);
        // Fill the left array with the counts of consecutive 1's from the left
        for (int i = 1; i < size; ++i) {</pre>
            if (nums[i - 1] == 1) {
                left[i] = left[i - 1] + 1;
        // Fill the right array with the counts of consecutive 1's from the right
        for (int i = size - 2; i >= 0; --i) {
            if (nums[i + 1] == 1) {
                right[i] = right[i + 1] + 1;
        // Initialize the variable to store the maximum length of the subarray
        int max_length = 0;
        // Iterate over the input array to compute the maximum subarray length
        for (int i = 0; i < size; ++i) {
            // The longest subarray is the sum of consecutive 1's to the left and right of the current element
            max_length = max(max_length, left[i] + right[i]);
        // Return the computed maximum subarray length
        return max_length;
};
TypeScript
function longestSubarray(nums: number[]): number {
    // Get the size of the input array
```

// should not be equal to the total length of the array (which implies no 0 was in the array to begin with).

// Return the computed maximum subarray length return maxLength; from typing import List

•

•

Space Complexity:

as the input array nums.

class Solution:

n = len(nums)

```
# Initialize two lists to keep track of consecutive ones to the left and right of each index
        left ones count = [0] * n
        right_ones_count = [0] * n
        # Calculate the consecutive ones to the left of each index
        for i in range(1, n):
            if nums[i - 1] == 1:
                left_ones_count[i] = left_ones_count[i - 1] + 1
        # Calculate the consecutive ones to the right of each index
        for i in range(n - 2, -1, -1):
            if nums[i + 1] == 1:
                right_ones_count[i] = right_ones_count[i + 1] + 1
        # Find the maximum length subarray formed by summing up counts of left and right ones.
        # Note that the question assumes we can remove one zero to maximize the length.
        # So, connecting two streaks of ones effectively means removing one zero between them.
        max_length = max(a + b for a, b in zip(left_ones_count, right_ones_count))
        return max_length
Time and Space Complexity
  The given code aims to find the length of the longest subarray of 1s after deleting one element from the array. Here's the analysis
  of its computational complexity:
Time Complexity:
      The time complexity of this algorithm is O(n), where n is the length of the input array nums. The reasoning behind this is that
      there are two separate for loops that each iterate over the array exactly once. The first loop starts from index 1 and goes up
      to n-1, incrementing by 1 on each iteration. The second loop starts from n-2 and goes down to 0, decrementing by 1 on each
```

iteration. In both of these loops, only a constant amount of work is done (simple arithmetic and array accesses), so the time

Furthermore, there's a final step that combines the results of the left and right arrays using max and zip, which is also a

linear operation since it involves a single pass over the combined arrays, making it O(n). Adding up these linear time operations (3 * 0(n)) still results in an overall time complexity of 0(n).

complexity for each loop is linear with respect to the size of the array.

The space complexity of this algorithm is O(n) as well. This is because two additional arrays, left and right, are created to store the lengths of continuous ones to the left and right of each index, respectively. Each of these arrays is the same length

- Besides the two arrays left and right, there are only a constant number of variables used (i.e., n, i, a, b), so they do not add more than 0(1) to the space complexity.
- Thus, the total space complexity is determined by the size of the two additional arrays, which gives us O(n).