2134. Minimum Swaps to Group All 1's Together II

Problem Description

Medium <u>Array</u>

Sliding Window

The problem provides us with a binary circular array, which means the elements can be either 0 or 1, and the first and last elements are considered adjacent. Our task is to find the minimum number of swaps needed to group all the 1s together at any location in the array. A swap is defined as taking two distinct positions in an array and exchanging the values in them.

To better understand the concept of a circular array, imagine that the last element in the array is followed by the first element,

thereby creating a loop. This loop-like behavior suggests that the grouping of 1s can span from the end of the array to the beginning.

The goal is to achieve this grouping using the fewest number of swaps, and we need to return that number.

Intuition To solve this problem, we need to focus on the group of 1s and how many 0s we find within the windows that can cover all 1s.

counts of 1s up to each index so that we can calculate the count of 1s within any range in constant time.

that involve contiguous subarrays. Here's a step-by-step breakdown of the implementation:

Since the array is circular, these windows can wrap around the end to the start of the array.

We begin by counting the total number of 1s in the array (cnt). Knowing this count allows us to determine the size of the window we'll be sliding across the array. The size of the window is equal to the total number of 1s since our objective is to cluster all 1s together.

We then double the size of the array to simplify the handling of the circular nature of the problem. By doing this, we avoid the need for modular arithmetic and can work with a simple sliding window technique on this extended array. To efficiently check how many 1s are within any window, we construct a prefix sum array (s). This array holds the cumulative

With the prefix sum array in hand, we then slide a window of size cnt over the array and track the maximum number of 1s we can find within such a window (mx). The position of this window will vary, and it could span the end and beginning of the original array

due to its circular nature. Finally, the minimum number of swaps is determined by the size of the window (cnt) minus the maximum number of 1s found

within any window (mx). This is because if a window has x number of 1s, then it needs cnt - x swaps to move all 1s inside it

Solution Approach The solution approach hinges on the <u>sliding window</u> technique, which is a common pattern used to solve array-based problems

Count the Number of 1s: First, count the total number of 1s in the given array (cnt). This is done using the count method

Calculate the Extended Prefix Sum: Create an array s which will store the prefix sums of the array doubled in length. This

doubling accounts for the circular nature without dealing with wraparounds during the window sliding. If nums is [1,0,1,0,1],

end of the current window.

swapped with 1s outside of it.

Count the Number of 1s:

that index in the extended array.

the number of 1s in each window. For instance:

because for each 0 in the window, one swap is needed.

which is O(n) where n is the number of elements in the array.

then this step considers [1,0,1,0,1,1,0,1,0,1]. The prefix sum for each index i is calculated by adding s[i] to nums[i % n], storing the cumulative count of 1s up to index i.

- Slide the Window and Find Maximum 1s: Slide a window of length cnt (the number of 1s in the array) and find the maximum number of 1s in any such window in the extended array. The variable mx keeps track of this maximum count.
- Please note that the iteration for the sliding window is only until j < (n << 1) to ensure we do not access prefix sum array out of bounds. Here n << 1 is a bitwise operation equivalent to multiplying n by 2, which is the length of our extended array. Compute the Result: Once the maximum number of 1s in any window is determined (mx), the result is simply the window size

cnt minus mx. This is because the number of required swaps is equal to the number of 0s in the window that need to be

The beauty of this solution is that it reduces what could be a complex modular arithmetic problem (due to the circular nature) into

Iterate over each possible position the window can start from in the doubled array and compute the number of 1s in the

current window using the prefix sums - by doing a constant time operation s[j + 1] - s[i], where i is the start and j is the

Example Walkthrough Let's use a small example to illustrate the solution approach described above. Consider the binary circular array nums given by

a simpler linear problem by cleverly extending the size of the array and using a prefix sum array for efficient range queries.

[1,0,1,0,0]. We want to find the minimum number of swaps needed to group all the 1s together.

 \circ Starting at index 0: the window includes [1,0], and the count from the prefix sum is s[2] - s[0] = 1 - 0 = 1.

• ... and so on, until we cover all possible start positions of the window in the extended array.

Calculate the Extended Prefix Sum: We double the length of nums so that it becomes [1,0,1,0,0,1,0,1,0,0]. The prefix sum array s will be calculated from this

extended array: s = [0,1,1,2,2,2,3,3,4,4,4]. Notice how for each index i, s[i+1] accounts for the total number of 1s up to

Slide the Window and Find Maximum 1s: We slide a window of length 2 across the array and use the prefix sum to calculate

From these calculations, we find that the maximum number of 1s in any window of size cnt is 1 (in this case, since all windows

In nums, there are two 1s. So, cnt = 2. This means our sliding window will need to be of size 2 because we want all 1s to be

• Starting at index 1: the window includes [0,1], and the count from the prefix sum is s[3] - s[1] = 2 - 1 = 1. \circ Starting at index 2: the window includes [1,0], and the count from the prefix sum is s[4] - s[2] = 2 - 1 = 1.

contain only one 1).

Solution Implementation

length_of_nums = len(nums)

Python

together.

number of swaps required is cnt - mx = 2 - 1 = 1. Thus, only a single swap is necessary to group all 1s together. In the given example, we can swap the first 0 right after the last 1 to get [1,1,0,0,0], and all 1s are now grouped together using just one swap.

This example clearly demonstrates each step of the solution approach and how it can effectively minimize the problem

complexity, leveraging a sliding window mechanism on an extended array with the help of prefix sums.

Create an extended sum list which is twice the length of the nums list plus one

Fill the sum list with prefix sums, allowing wrap around to simulate a circular array

Iterate through the prefix_sum array to find the maximum number of ones in any subarray

Update max_ones_found with the maximum ones found in the current sliding window

// Compute the number of 1's in the current range and update maxOnes if necessary

of the size count_of_ones, which is the number of swaps needed on a circular array

This is for the purpose of creating a sliding window later on

prefix_sum[i + 1] = prefix_sum[i] + nums[i % length_of_nums]

prefix_sum = [0] * ((length_of_nums << 1) + 1)</pre>

for i in range(length_of_nums << 1):</pre>

for i in range(length_of_nums << 1):</pre>

Initialize max_ones_found variable as 0

end_index = i + count_of_ones - 1

if end_index < (length_of_nums << 1):</pre>

// Count how many 1's are there in the array

int[] sumArray = new int[(n << 1) + 1];</pre>

for (int i = 0; i < (n << 1); ++i) {

for (int i = 0; i < (n << 1); ++i) {

int j = i + onesCount - 1;

if (j < (n << 1)) {

print(result) # Output would be the minimum number of swaps required

Compute the Result: Given that cnt = 2 and the maximum number of 1s we found in any window (mx) is 1, the minimum

Import the List type from types module for type hinting from typing import List class Solution: def minSwaps(self, nums: List[int]) -> int: # Count the number of ones in the list count_of_ones = nums.count(1) # Get the length of the list

```
max_ones_found = max(max_ones_found, prefix_sum[end_index + 1] - prefix_sum[i])
# The minimum number of swaps needed is the total one count minus the maximum ones found
return count_of_ones - max_ones_found
```

result = sol.minSwaps([0,1,0,1,1,0,0])

public int minSwaps(int[] nums) {

for (int value : nums) {

onesCount += value;

int onesCount = 0;

int n = nums.length;

int maxOnes = 0;

Example usage:

class Solution {

Java

sol = Solution()

max_ones_found = 0

```
// The minimum number of swaps is the difference between total ones and the maximum ones found in any range of size ones(
return onesCount - maxOnes;
```

// Create an extended array of sums to handle circular array

// Determine the end index for the range of size onesCount

maxOnes = Math.max(maxOnes, sumArray[j + 1] - sumArray[i]);

sumArray[i + 1] = sumArray[i] + nums[i % n];

```
class Solution {
public:
    int minSwaps(vector<int>& nums) {
       // Count the total number of ones in the input vector
        int oneCount = 0;
        for (int value : nums) {
            oneCount += value;
        int n = nums.size();
       // Initialize an extended sum vector that is twice as long as the input
       // This is used to simulate a circular array
        vector<int> prefixSum((n << 1) + 1, 0);</pre>
        for (int i = 0; i < (n << 1); ++i) {
           // Populate the prefix sum array by adding the current element
           // Note: Use modulus to simulate the circular array
            prefixSum[i + 1] = prefixSum[i] + nums[i % n];
        int maxOnes = 0;
       // Slide a window of length equal to the number of ones (oneCount)
       // over the prefix sum array to find the maximum number of ones in any
        // subarray of the same length
        for (int i = 0; i < (n << 1); ++i) {
           // Calculate the end of the window
            int windowEnd = i + oneCount - 1;
           // Ensure that we do not go past the end of the sum array
            if (windowEnd < (n << 1)) {</pre>
                int windowSum = prefixSum[windowEnd + 1] - prefixSum[i];
                maxOnes = max(maxOnes, windowSum);
       // The minimum number of swaps needed is the difference between
       // the number of ones in the array and the maximum number of ones
       // found in any window of size equal to the number of ones.
       // This tells us how many zeros we need to swap out of the window.
        return oneCount - max0nes;
TypeScript
```

```
# Import the List type from types module for type hinting
from typing import List
class Solution:
   def minSwaps(self, nums: List[int]) -> int:
```

};

function minSwaps(nums: number[]): number {

// Get the length of the input array.

return totalOnes - maxOnesInAnyWindow;

Count the number of ones in the list

for i in range(length_of_nums << 1):</pre>

Initialize max_ones_found variable as 0

prefix_sum = [0] * ((length_of_nums << 1) + 1)</pre>

count_of_ones = nums.count(1)

Get the length of the list

length_of_nums = len(nums)

max_ones_found = 0

Time and Space Complexity

const arrayLength: number = nums.length;

// Calculate the total number of 1's in the array.

let maxOnesInAnyWindow: number = onesCountInWindow;

for (let i = totalOnes; i < arrayLength + totalOnes; i++) {</pre>

let elementEntering: number = nums[i % arrayLength];

onesCountInWindow += elementEntering - elementExiting;

// Identify the element going out of the window.

// Identify the new element entering the window.

// Update the count of 1's in the current window.

const totalOnes: number = nums.reduce((acc, current) => acc + current, 0);

// Iterate over each window of size 'totalOnes' in the circular array.

let elementExiting: number = nums[(i - totalOnes) % arrayLength];

maxOnesInAnyWindow = Math.max(onesCountInAnyWindow, maxOnesInAnyWindow);

Create an extended sum list which is twice the length of the nums list plus one

Fill the sum list with prefix sums, allowing wrap around to simulate a circular array

This is for the purpose of creating a sliding window later on

prefix_sum[i + 1] = prefix_sum[i] + nums[i % length_of_nums]

// Initialize the count of 1's in the first window of size equal to total number of 1's.

let onesCountInWindow: number = nums.slice(0, totalOnes).reduce((acc, current) => acc + current, 0);

// Update the maximum count of 1's found so far in any window if current window has more.

// Calculate minimum swaps as total number of 1's minus the maximum number of 1's in a window.

// Initialize the maximum count of 1's found in any window — this will be used to calculate minimum swaps.

```
# Iterate through the prefix_sum array to find the maximum number of ones in any subarray
        # of the size count_of_ones, which is the number of swaps needed on a circular array
        for i in range(length_of_nums << 1):</pre>
            end_index = i + count_of_ones - 1
            if end_index < (length_of_nums << 1):</pre>
                # Update max_ones_found with the maximum ones found in the current sliding window
                max ones found = max(max ones found, prefix sum[end index + 1] - prefix sum[i])
        # The minimum number of swaps needed is the total one count minus the maximum ones found
        return count_of_ones - max_ones_found
# Example usage:
# sol = Solution()
\# result = sol.minSwaps([0,1,0,1,1,0,0])
# print(result) # Output would be the minimum number of swaps required
```

Populating the s array requires iterating over each element once, and since it's done over 2 * n elements (to handle the

Time Complexity

circular nature of the problem), this operation has a time complexity of O(2n), which simplifies to O(n). The calculation of mx within the second loop involves iterating up to 2 * n times, and within each iteration, we perform a

constant time operation of computing the sum and finding the maximum. This results in a time complexity of O(2n), again

The time complexity of the given code is mainly determined by two loops: the loop used to populate the s array and the loop

- simplifying to O(n). Overall, since both operations are sequential and not nested, the total time complexity of the code is O(n).
- **Space Complexity**

The s array is the primary additional data structure, which has a length of (2 * n) + 1. Therefore, the space complexity due

The space complexity is determined by the additional space required besides the input. In this algorithm:

Other variables used (i, j, mx, cnt) are all constant-sized, adding a negligible 0(1) to the space complexity. • Hence, the total space complexity of the algorithm is O(n).

to s alone is O(2n), which simplifies to O(n).

used to find the maximum number of 1's within a window of size cnt.