1151. Minimum Swaps to Group All 1's Together

Sliding Window Medium <u>Array</u>

Problem Description

1s together in the array. The 1s can be placed at any location within the array, but they must be consecutive (i.e., no 0s in between them). By swap, we mean taking two adjacent elements and exchanging their positions. This problem is asking us to perform a series of swaps to bring all 1s together and minimize the number of swaps made in this process.

Given a binary array data, which consists only of 0s and 1s, the task is to find the minimum number of swaps required to group all

To provide more clarity, a swap can only be made with two adjacent elements, which means you cannot take a 1 and swap it with

you minimize the total number of swaps required. Intuition

another 1 if there is a 0 in between without first swapping that 0 out of the way. The goal is to find the optimal strategy so that

Firstly, we must realize that we are grouping all 1s together. Instead of focusing on the 0s we need to move, we concentrate on

for grouping 1s together. The size of this segment (i.e., the window) will be equal to the total number of 1s in the array, let's call it

subtract the element that leaves the window (data[i - k]) from t.

number of 0s in the window that contains the maximum number of 1s.

give us the minimum number of 0s to swap with.

the window (1), so t becomes 1. mx remains 2.

group all 1s together in this example.

total_ones = data.count(1)

max_ones = current_count

current_count = sum(data[:total_ones])

for i in range(total_ones, len(data)):

current_count -= data[i - total_ones]

max_ones = max(max_ones, current_count)

current_count += data[i]

return total_ones - max_ones

public int minSwaps(int[] data) {

number of 1s we found together in all windows of size k=4 was 2.

Let's go through the solution approach with a small example using the given binary array data:

The intuition behind the solution involves sliding window and greedy strategies.

k. So, the window will be a subarray of length k. The number of swaps required will then be the number of 0s in our optimal segment because each of these 0s will need to be swapped with a 1 outside the segment. To find the optimal segment, we consider all possible segments of length k by using a sliding window through the array. For each

the segment containing the maximum number of 1s. Why? Because the fewer the 0s in this segment, the fewer swaps are needed

segment, we calculate how many 1s it contains — this tells us indirectly the number of 0s since the window's length is fixed. We want the segment with the maximum number of 1s because that would mean the minimum number of 0s, and thus the minimum number of swaps needed.

this first window. It then moves the window across the array one element at a time, calculating the count of 1s by adding data[i] and subtracting data[i - k] which keeps the window size constant. If the count is greater than mx, it updates mx. At last, because k is the total number of 1s which should be grouped together, the minimum number of swaps needed is equal to the number of 0s in the optimal segment, which is k - mx.

The code sets up this sliding window starting from the beginning of the array and initializes a variable mx with the number of 1s in

Solution Approach The solution leverages the [sliding window] (/problems/sliding_window_maximum) technique to efficiently compute the number of 1s in each window of size k, where k is the total number of 1s in the input array data. Here is a walkthrough of the implementation steps:

Counting 1s in the Array: We start by counting the number of 1s in the entire array using data.count(1), which will determine

The result gives us the number of 1s in the initial window, which is stored in the variable t. We also introduce a variable mx to

the size of our <u>sliding window</u>. This count is stored in the variable k. Initial Window Setup: We set up the initial window by calculating the sum of the first k elements in the array (sum(data[:k])).

keep track of the maximum number of 1s found in any window, which is initially equal to t. Sliding Window Movement: We then iterate through the array starting from the kth element. In each iteration, we simulate

the sliding of the window by one element to the right. We add the new element that enters the window (data[i]) to t and

Updating the Maximum 1s Count: After adjusting t for the new window position, we check if the updated count is greater than the current maximum (mx). If it is, we update mx to this new value. Calculating the Result: Once we have completed sliding through the array, we subtract the maximum 1s count (mx) from the

total number of 1s (k). The result (k - mx) represents the minimum number of swaps required to group all 1s because it's the

visited at most twice - once when it enters the window and once when it leaves. **Example Walkthrough**

The time complexity of this solution is O(n), where n is the length of the input array. This is because each element in data is

Initial Window Setup: We set up the initial window over the first 4 elements: 0 1 0 1. The count of 1s in this window is t = 2. We also initialize mx with the value of t, so mx = 2.

subtracting the first element of the previous window (0), so t remains 2. mx does not change.

Counting 1s in the Array: There are 4 1s in the array, so k = 4. We are looking for a contiguous segment of size k which will

Move the window one step right to 1 0 1 0. Update t by adding the new element (0) which is entering the window and

Move the window one step right to 0 1 0 1. Update t by adding the new element (1) and subtracting the element that left the window (1), so t remains 2. mx is still 2.

Sliding Window Movement:

Example data array:

0 1 0 1 0 1 0 0 1

- Move the window one step right to 1 0 1 0. Update t by adding the new element (0) and subtracting the element that left the window (0), so t remains 2. mx is still 2.
 - Finally, move the window to the last possible position 1 0 0 1. Update t by adding the new element (1) and subtracting the element that left the window (0), t becomes 2. mx remains 2 since it's not less than 2.

Updating the Maximum 1s Count: During the above window movements, mx never increased above 2, so the maximum

Calculating the Result: The minimum number of swaps required is k - mx = 4 - 2 = 2. So, we need at least 2 swaps to

In summary, using the sliding window technique makes us move across the array, updating the number of 1s in each window and

Move the window one step right to 0 1 0 0. Update t by adding the new element (0) and subtracting the element that left

minimum number of swaps needed. **Solution Implementation**

keeping track of the maximum. The final answer is derived from the size of our window minus this maximum value, giving the

from typing import List class Solution: def min_swaps(self, data: List[int]) -> int: # Calculate the total number of 1s needed to form a continuous subarray.

Include the next element in the window and remove the trailing element to slide the window forward.

The minimum number of swaps equals the number of 0s in the largest window of 1s (size of the window — max count of 1s).

Initialize the current count of 1s in the first window of size 'total_ones'.

// Count the number of 1's in the array, which will be the window size 'k'

Initialize the maximum count of 1s found so far to the current count of the initial window.

Iterate over the array starting from the end of the first window to the end of the array.

Update the maximum count of 1s if the current window has more 1s than any previous ones.

Example usage: # solution = Solution() # print(solution.min_swaps([1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1])) # Output: 3

Java

C++

public:

class Solution {

class Solution {

Python

```
int numOfOnes = 0;
for (int value : data) {
    numOfOnes += value;
// Initialize the total number of 1's in the first window of size 'k'
int onesInCurrentWindow = 0;
for (int i = 0; i < numOfOnes; ++i) {</pre>
    onesInCurrentWindow += data[i];
// This variable will keep track of the maximum number of 1's
// found in any window of size 'k'
int maxOnesInWindow = onesInCurrentWindow;
// Slide the window of size 'k' through the array while updating the number
// of 1's in the current window and the maximum found so far
for (int i = numOfOnes; i < data.length; ++i) {</pre>
    // Include the next element in the window
    onesInCurrentWindow += data[i];
    // Exclude the first element of the previous window
    onesInCurrentWindow -= data[i - numOfOnes];
    // Update the maximum number of 1's in any window if the current window
    // has more 1's
    maxOnesInWindow = Math.max(maxOnesInWindow, onesInCurrentWindow);
// The minimum number of swaps is the size of the window 'k' minus
// the maximum number of 1's that can be placed in a window
return numOfOnes - maxOnesInWindow;
```

```
};
TypeScript
```

// and subtracting the element that's no longer in the window.

maxOnesInWindow = Math.max(maxOnesInWindow, currentOnes);

// Count the number of 1s in the array, which is the window size we're interested in (k).

// Update the count of 1s in the current window by adding the new incoming element

// Update the maximum number of 1s in any window if the current window has more.

// Track the maximum number of 1s found in any window of size k (to minimize swaps).

const totalOnes = data.reduce((accumulator, currentValue) => accumulator + currentValue, 0);

let currentOnes = data.slice(0, totalOnes).reduce((accumulator, currentValue) => accumulator + currentValue, 0);

// The minimum number of swaps is the difference between the total

// number of 1's and the maximum number of 1's in a window

#include <vector> // Include the vector header for using std::vector

// all the 1's together in the array.

int windowSize = 0;

int oneCount = 0;

for (int value : data) {

windowSize += value;

oneCount += data[i];

oneCount += data[i];

return windowSize - max0nes;

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

let max0nesInWindow = currentOnes;

int maxOnes = oneCount;

for (int i = 0; i < windowSize; ++i) {

int minSwaps(std::vector<int>& data) {

#include <algorithm> // Include the algorithm header for the std::max function

// This function calculates the minimum number of swaps required to group

// Count the number of 1's in the array, which will be the window size

// Count the number of 1's in the initial window of size 'windowSize'

// Slide the window across the array to find the maximum number of 1's

// Initialize the maximum number of 1's found in a window

// that can be contained in a window of size 'windowSize'

// Exclude the oldest element from the window

for (int i = windowSize; i < data.size(); ++i) {</pre>

// Include the new element in the window

// Update the maximum number of 1's found

maxOnes = std::max(maxOnes, oneCount);

// Initial count of 1s in the first window of size k.

// Slide the window of size k through the data array.

currentOnes += data[i] - data[i - totalOnes];

for (let i = totalOnes; i < data.length; ++i) {</pre>

oneCount -= data[i - windowSize];

```
// The minimum number of swaps equals the window size minus the maximum number
      // of 1s in any window, which tells us how many 0s need to be swapped out.
      return totalOnes - maxOnesInWindow;
from typing import List
class Solution:
   def min_swaps(self, data: List[int]) -> int:
       # Calculate the total number of 1s needed to form a continuous subarray.
        total ones = data.count(1)
       # Initialize the current count of 1s in the first window of size 'total_ones'.
        current_count = sum(data[:total_ones])
       # Initialize the maximum count of 1s found so far to the current count of the initial window.
       max_ones = current_count
       # Iterate over the array starting from the end of the first window to the end of the array.
        for i in range(total_ones, len(data)):
            # Include the next element in the window and remove the trailing element to slide the window forward.
            current_count += data[i]
            current count -= data[i - total ones]
            # Update the maximum count of 1s if the current window has more 1s than any previous ones.
            max_ones = max(max_ones, current_count)
       # The minimum number of swaps equals the number of 0s in the largest window of 1s (size of the window — max count of 1s).
        return total ones - max ones
```

Time Complexity: The time complexity of the minSwaps function is O(n), where n is the length of the data array. The function comprises of two main

print(solution.min_swaps([1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1])) # Output: 3

Example usage:

solution = Solution()

Time and Space Complexity

operations: Counting the number of 1s in the data array using data.count(1). This operation goes through each element of the array,

Combining both parts, the overall time complexity is O(n) + O(n) = O(n).

resulting in a time complexity of O(n). The sliding window loop, which starts from index k up to the end of the array. In each iteration, the function adds the current

constant time operations, the time complexity of the loop is 0(n - k). Since k is less than or equal to n, the loop still implies an O(n) time complexity.

element and subtracts the element k positions before it. The loop runs n - k times. Since addition and subtraction are

Space Complexity: The space complexity of this function is 0(1). It uses a fixed number of variables (k, t, and mx) that do not depend on the size of

the input. No additional data structures that scale with the size of the input are used. The variables i used for iteration and the space required for data.count(1) are also constant and do not contribute to the space complexity beyond 0(1).