1151. Minimum Swaps to Group All 1's Together

Medium Array Sliding Window

Leetcode Link

Problem Description

Given a binary array `data`, which consists only of `0`s and `1`s, the task is to find the minimum number of swaps required to group all `1`s together in the array. The `1`s can be placed at any location within the array, but they must be consecutive (i.e., no `0`s 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 `1`s together and minimize the number of swaps made in this process.

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 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 you minimize the total number of swaps required.

Intuition

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

Firstly, we must realize that we are grouping all `1`s together. Instead of focusing on the `0`s we need to move, we concentrate on the segment containing the maximum number of `1`s. Why? Because the fewer the `0`s in this segment, the fewer swaps are needed for grouping `1`s together. The size of this segment (i.e., the window) will be equal to the total number of `1`s in the array, let's call it k. So, the window will be a subarray of length k.

The number of swaps required will then be the number of `0`s in our optimal segment because each of these `0`s 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 segment, we calculate how many `1`s it contains — this tells us indirectly the number of `0`s since the window's length is fixed. We want the segment with the maximum number of `1`s because that would mean the minimum number of `0`s, and thus the minimum number of swaps needed.

The code sets up this sliding window starting from the beginning of the array and initializes a variable `mx` with the number of `1`s in this first window. It then moves the window across the array one element at a time, calculating the count of `1`s 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 `1`s which should be grouped together, the minimum number of swaps needed is equal to the number of `0`s in the optimal segment, which is $k - mx$.

Solution Approach

The solution leverages the `sliding window` technique to efficiently compute the number of `1`s in each window of size `k`, where `k` is the total number of `1`s in the input array `data`.

Here is a walkthrough of the implementation steps:

1. **Counting `1`s in the Array:** We start by counting the number of `1`s in the entire array using `data.count(1)`, which will determine the size of our sliding window. This count is stored in the variable `k`.
2. **Initial Window Setup:** We set up the initial window by calculating the sum of the first `k` elements in the array (`sum(data[:k])`). The result gives us the number of `1`s in the initial window, which is stored in the variable `t`. We also introduce a variable `mx` to keep track of the maximum number of `1`s found in any window, which is initially equal to `t`.
3. **Sliding Window Movement:** We then iterate through the array starting from the `k`th 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 subtract the element that leaves the window (`data[i - k]`) from `t`.
4. **Updating the Maximum `1`s 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.
5. **Calculating the Result:** Once we have completed sliding through the array, we subtract the maximum `1`s count (`mx`) from the total number of `1`s (`k`). The result (`k - mx`) represents the minimum number of swaps required to group all `1`s because it's the number of `0`s in the window that contains the maximum number of `1`s.

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

Example Walkthrough

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

Example `data` array:

```
1 0 1 0 1 0 1
```

1. **Counting `1`s in the Array:** There are 4 `1`s in the array, so $k = 4$. We are looking for a contiguous segment of size k which will give us the minimum number of `0`s to swap with.
2. **Initial Window Setup:** We set up the initial window over the first 4 elements: `0 1 0 1`. The count of `1`s in this window is $t = 2$. We also initialize mx with the value of t, so $mx = 2$.
3. **Sliding Window Movement:**
   - 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 subtracting the first element of the previous window (`0`), so t remains 2. mx does not change.
   - 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.
   - 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.
   - 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 remains 2.
   - 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 becomes 1. mx remains 2.
   - Finally, move the window to the last possible position `1 0 1 0`. Update t by adding the new element (`0`) and subtracting the element that left the window (`1`), so t becomes 2. mx remains 2 since it's not less than 2.
4. **Updating the Maximum `1`s Count:** During the above window movements, mx never increased above 2, so the maximum number of `1`s we found together in all windows of size `4` was 2.
5. **Calculating the Result:** The minimum number of swaps required is $k - mx = 4 - 2 = 2$. So, we need at least 2 swaps to group all `1`s together in this example.

In summary, using the sliding window technique makes us move across the array, updating the number of `1`s in each window and keeping track of the maximum. The final answer is derived from the size of our window minus this maximum value, giving the minimum number of swaps needed.

Python Solution

```python
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

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

Java Solution

```java
class Solution {
    public int minSwaps(int[] data) {
        // Count the number of 1's in the array, which will be the window size 'k'
        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) {
            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) {
            // 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;
    }
}
```

C++ Solution

```cpp
#include <vector>  // Include the vector header for using std::vector
#include <algorithm> // Include the algorithm header for the std::max function

class Solution {
public:
    // This function calculates the minimum number of swaps required to group
    // all the 1's together in the array.
    int minSwaps(std::vector<int>& data) {
        // Count the number of 1's in the array, which will be the window size
        int windowSize = 0;
        for (int value : data) {
            windowSize += value;
        }

        // Count the number of 1's in the initial window of size 'windowSize'
        int oneCount = 0;
        for (int i = 0; i < windowSize; ++i) {
            oneCount += data[i];
        }

        // Initialize the maximum number of 1's found in a window
        int maxOnes = oneCount;

        // Slide the window across the array to find the maximum number of 1's
        // that can be contained in a window of size 'windowSize'
        for (int i = windowSize; i < data.size(); ++i) {
            // Include the new element in the window
            oneCount += data[i];
            // Exclude the oldest element from the window
            oneCount -= data[i - windowSize];
            // Update the maximum number of 1's found
            maxOnes = std::max(maxOnes, oneCount);
        }

        // 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
        return windowSize - maxOnes;
    }
};
```

Typescript Solution

```typescript
function minSwaps(data: number[]): number {
    // Count the number of 1s in the array, which is the window size we're interested in (k).
    const totalOnes = data.reduce((accumulator, currentValue) => accumulator + currentValue, 0);

    // Initial count of 1s in the first window of size k.
    let currentOnes = data.slice(0, totalOnes).reduce((accumulator, currentValue) => accumulator + currentValue, 0);

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

    // Slide the window of size k through the data array.
    for (let i = totalOnes; i < data.length; ++i) {
        // Update the count of 1s in the current window by adding the new incoming element
        // and subtracting the element that's no longer in the window.
        currentOnes += data[i] - data[i - totalOnes];

        // Update the maximum number of 1s in any window if the current window has more.
        maxOnesInWindow = Math.max(maxOnesInWindow, currentOnes);
    }

    // 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;
}
```

Time and Space Complexity

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 operations:

1. Counting the number of `1`s in the `data` array using `data.count(1)`. This operation goes through each element of the array, resulting in a time complexity of `O(n)`.
2. The sliding window loop, which starts from index `k` up to the end of the array. In each iteration, the function adds the current element and subtracts the element `k` positions before it. The loop runs `n - k` times. Since addition and subtraction are constant time operations, the time complexity of the loop is `O(n - k)`. Since `k` is less than or equal to `n`, the loop still implies an `O(n)` time complexity.

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

Space Complexity:

The space complexity of this function is `O(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 `O(1)`.