1664. Ways to Make a Fair Array

Medium Array Dynamic Programming Leetcode Link

Problem Description

In this problem, you are given an array of integers called `nums`. Your task is to determine the number of ways you can remove exactly one element from the array such that, after the removal, the sum of the elements at the odd indices is equal to the sum of the elements at the even indices. The array uses 0-based indexing, which means the first element is at index 0 (even-indexed), the second element is at index 1 (odd-indexed), and so on.

Imagine you have an array like `[2, 1, 3, 4]`. If you remove the element at index 1 (which is the number `1`), the new array would be `[2, 3, 4]`. In this new array, the sum of the even-indexed values (which are at indices 0 and 2) is 2 + 4 = 6, and the sum of the odd-indexed value (which is at index 1) is 3. Since those sums are not equal, this would not be a "fair" array according to the problem definition.

Intuition

The intuition behind the solution is to efficiently track the sums of odd and even indexed elements as we consider removing each element. If we just recalculated the sums each time we remove an element, it would be too slow, so we need a smarter approach.

We start by calculating the sum of all elements at even indices (`s1`) and the sum of all elements at odd indices (`s2`) for the original array. Once we have the total sums, as we iterate through each element to consider its removal, we can update our totals for what the sums would be if we removed that element.

As we remove an element, two scenarios can occur depending on whether the element is at an even or odd index:

1. If the element is at an even index, we need to subtract that element's value from the even sum and add it to the odd sum (since all elements to the right will shift left by one index).
2. If the element is at an odd index, we need to subtract that element's value from the odd sum and add it to the even sum.

We keep track of the running tallies of elements removed (`t1` for even indices and `t2` for odd indices). To avoid shifting all elements and updating all sums every time, we make adjustments based on the running tallies and the totals.

We then check if removing the current element makes the sum of even-indexed elements equal to the sum of odd-indexed elements. If so, we increment our answer count `ans`. The condition checks depend on the parity of the index and use the already computed sums as well as the tallies for correction.

This way, we are able to efficiently process each element and determine if its removal results in a "fair" array without having to recalculate the entire sum each time.

Solution Approach

The implementation of the solution revolves around the use of prefix sums, which is a common technique in array manipulation problems, to pre-calculate cumulating values and use them in an efficient way. The solution makes use of several variables: `s1` and `s2` to keep track of the sum of elements at even indices and odd indices respectively, `ans` to count the number of ways we can make the array fair, and `t1` and `t2` to keep track of the prefix sums (total of removed elements so far) at even and odd indices respectively.

The algorithm consists of the following steps:

1. Calculate the initial sums of even and odd indexed elements and store them in `s1` and `s2`.

```python
s1, s2 = sum(nums[::2]), sum(nums[1::2])
```

2. Initialize `ans`, `t1`, and `t2` to `0`.
3. Loop through each element in the `nums` array using their index `i` and value `v`.
4. Inside the loop, perform checks to determine if removing the element at index `i` will result in a fair array:
   - Check if the index is even (`i % 2 == 0`). If true, compare `t2 + s1 - t1 - v` with `t1 + s2 - t2`. This check is based on the idea that, after removal, the total sum of even-indexed elements (`s1`) would be decreased by `v`, and the total sum of odd-indexed elements (`s2`) would need to include what was previously part of `t1` before the `i-th` element. If the two sums are equal, there is a valid removal, so increment `ans`.
   - Check if the index is odd (`i % 2 == 1`). If true, compare `t2 + s1 - t1` with `t1 + s2 - t2 - v`. Similar to the previous case, but in this scenario, we're removing an element from the odd indices, so we adjust `s2` instead of `s1`, and compare against the sum of even-indexed elements plus the prefix sum `t2`.

```python
for i, v in enumerate(nums):
    ans += i % 2 == 0 and t2 + s1 - t1 - v == t1 + s2 - t2
    ans += i % 2 == 1 and t2 + s1 - t1 == t1 + s2 - t2 - v
    t1 += v if i % 2 == 0 else 0
    t2 += v if i % 2 == 1 else 0
```

5. Update `t1` and `t2` with the value `v` pertaining to their respective index parities. This reflects the running sum of values that would have been removed up to index `i`.
6. After the loop, `ans` will contain the total number of indices that could be removed to make `nums` fair.

The time complexity of this solution is O(n) since it makes a single pass through the array, and the space complexity is O(1), using only a constant amount of extra space for the variables defined.

Example Walkthrough

Let's walk through an example to illustrate the solution approach using the array `[2, 1, 3, 4, 0]`.

1. We first calculate the initial sums of even and odd indexed elements:
   - `s1` = sum of elements at even indices = `2 + 3 + 0` = 5
   - `s2` = sum of elements at odd indices = `1 + 4` = 5
2. We initialize `ans`, `t1`, and `t2` to `0`.
3. We start to loop through each element in the `nums` array.
4. For `i = 0` (first element, value `v = 2`):
   - This is an even index, so we check if `t2 + s1 - t1 - v` equals `t1 + s2 - t2`.
   - That is, we check if `0 + 5 - 0 - 2` equals `0 + 5 - 0`.
   - The condition is `3 == 5`, which is false, so we do not increment `ans`.
   - We then update `t1` with the value `2` (since it's an even index), so `t1` becomes `2`, and `t2` remains `0`.
5. For `i = 1` (second element, value `v = 1`):
   - This is an odd index, so we compare `t2 + s1 - t1` with `t1 + s2 - t2 - v`.
   - That is, we check if `0 + 5 - 2` equals `2 + 5 - 0 - 1`.
   - The condition is `3 == 6`, which is false, so we do not increment `ans`.
   - We then update `t2` with the value `1` (since it's an odd index), so `t2` becomes `1`, and `t1` remains `2`.
6. For `i = 2` (third element, value `v = 3`):
   - This is an even index, so the comparison is `1 + 5 - 2 - 3` on the left side and `2 + 5 - 1` on the right side.
   - The condition is `1 == 6`, which is false, so `ans` stays the same.
   - We update `t1` to `5`, adding the value `3` to the previous value of `2`.
7. Continuing with this process for `i = 3` and `i = 4`, we find that:
   - At `i = 4`, with value `v = 0`, given as an even index, the condition would be `1 + 5 - 5 - 0` on the left side and `5 + 5 - 1` on the right side, which simplifies to `1 == 9`. The condition is false, so `ans` is still not incremented.
8. After the loop is completed, `ans` contains the value `0`, which indicates that there are no ways to remove a single element such that the sum of the even-indexed elements is equal to the sum of the odd-indexed elements.

Through this example, we can see that we process the array efficiently without recalculating the entire sum for every potential removal. This approach yields the correct answer with a linear time complexity.

Python Solution

```python
class Solution:
    def waysToMakeFair(self, nums: List[int]) -> int:
        # Calculate the initial sum of elements at even indices (odd positions) and
        # the sum of elements at odd indices (even positions)
        sum_even_index, sum_odd_index = sum(nums[::2]), sum(nums[1::2])

        # Initialize counters for the number of ways to make the array fair,
        # the running sum of elements at even indices, and odd indices
        fair_ways_count = running_sum_even = running_sum_odd = 0

        # Enumerate over the list to consider removing each element in turn
        for index, value in enumerate(nums):
            # If the current index is even, check if removing the element makes the array fair
            if index % 2 == 0:
                # A fair array has equal sum of remaining elements at even and odd positions
                is_fair_after_removal = (running_sum_odd + sum_even_index - running_sum_even - value ==
                                         running_sum_even + sum_odd_index - running_sum_odd)
                # If fair, increment the fair ways count
                fair_ways_count += is_fair_after_removal
            else:
                # If the current index is odd, perform a similar check after removing the element
                is_fair_after_removal = (running_sum_odd + sum_even_index - running_sum_even ==
                                         running_sum_even + sum_odd_index - running_sum_odd - value)
                fair_ways_count += is_fair_after_removal

            # Update the running sums for even and odd indices after considering each element
            if index % 2 == 0:
                running_sum_even += value
            else:
                running_sum_odd += value

        # Return the total number of ways the array can be made fair
        return fair_ways_count

```

Java Solution

```java
class Solution {
    public int waysToMakeFair(int[] nums) {
        int evenSum = 0; // Sum of elements at even indices
        int oddSum = 0;  // Sum of elements at odd indices
        int n = nums.length;

        // Calculate the initial sum of even and odd indexed numbers
        for (int i = 0; i < n; ++i) {
            if (i % 2 == 0) {
                evenSum += nums[i];
            } else {
                oddSum += nums[i];
            }
        }

        int tempEvenSum = 0; // Temporary sum for even indices
        int tempOddSum = 0;  // Temporary sum for odd indices
        int fairCount = 0;

        // Check each index to see if removing it would make the array fair
        for (int i = 0; i < n; ++i) {
            int currentValue = nums[i];

            // If the index is even, removing it would change the balance of sums
            if (i % 2 == 0) {
                // Check if the sums minus the current value are equal
                if (tempOddSum + evenSum - tempEvenSum - currentValue == tempEvenSum + oddSum - tempOddSum) {
                    fairCount++;
                }
                // Update the temporary sum for the even indices
                tempEvenSum += currentValue;
            } else {
                // Check if the sums minus the current value are equal
                if (tempOddSum + evenSum - tempEvenSum == tempEvenSum + oddSum - tempOddSum - currentValue) {
                    fairCount++;
                }
                // Update the temporary sum for the odd indices
                tempOddSum += currentValue;
            }
        }

        return fairCount; // Return result with the number of ways to make the array fair
    }
}

```

C++ Solution

```cpp
#include <vector>

class Solution {
public:
    int waysToMakeFair(std::vector<int>& nums) {
        int sumEven = 0, sumOdd = 0; // Initialize sums of even and odd indices
        int n = nums.size();

        // Calculate the total sum of elements at even and odd indices
        for (int i = 0; i < n; ++i) {
            if (i % 2 == 0) {
                sumEven += nums[i];
            } else {
                sumOdd += nums[i];
            }
        }

        int prefixSumEven = 0, prefixSumOdd = 0; // Prefix sums for even and odd indices
        int countFairIndices = 0; // This will hold the result - number of fair indices

        // Loop to find all indices where removing the element would make the array fair
        for (int i = 0; i < n; ++i) {
            int currentValue = nums[i];
            if (i % 2 == 0) {
                // Check if removing an element from even index makes sums equal
                if (prefixSumOdd + (sumEven - prefixSumEven - currentValue) == (prefixSumEven + sumOdd - prefixSumOdd)) {
                    countFairIndices++;
                }
                prefixSumEven += currentValue; // Update prefix sum for even index
            } else {
                // Check if removing an element from odd index makes sums equal
                if ((prefixSumOdd + sumEven - prefixSumEven) == (prefixSumEven + sumOdd - prefixSumOdd - currentValue)) {
                    countFairIndices++;
                }
                prefixSumOdd += currentValue; // Update prefix sum for odd index
            }
        }
        return countFairIndices; // Return the result
    }
};

```

Typescript Solution

```typescript
/**
 * Calculates the number of ways to delete exactly one element from the
 * array so that the sum of the elements at the odd indices of the new array
 * is equal to the sum of the elements at the even indices of the new array.
 * @param {number[]} nums - The input array.
 * @return {number} - The number of ways to make the array fair.
 */
var waysToMakeFair = function(nums: number[]): number {
    // Variables to keep track of even and odd sums.
    let evenSumOriginal: number = 0;
    let oddSumOriginal: number = 0;
    let tempEvenSum: number = 0;
    let tempOddSum: number = 0;

    const length = nums.length;

    // Calculate initial sums for even and odd indices.
    for (let i = 0; i < length; ++i) {
        if (i % 2 === 0) {
            evenSumOriginal += nums[i];
        } else {
            oddSumOriginal += nums[i];
        }
    }

    let fairWaysCount = 0;

    // Iterate through the array and count the ways to make the array fair
    // by testing the condition after removing each element.
    for (let i = 0; i < length; ++i) {
        const value = nums[i];

        // When removing an element at an even index, check if the sum without that
        // element makes the array fair.
        if (i % 2 === 0 && tempOddSum + evenSumOriginal - tempEvenSum - value === tempEvenSum + oddSumOriginal - tempOddSum) {
            fairWaysCount += 1;
        }

        // When removing an element at an odd index, check if the sum without that
        // element makes the array fair.
        if (i % 2 === 1 && tempOddSum + evenSumOriginal - tempEvenSum === tempEvenSum + oddSumOriginal - tempOddSum - value) {
            fairWaysCount += 1;
        }

        // Update temporary even and odd sums with the current element value.
        tempEvenSum += i % 2 === 0 ? value : 0;
        tempOddSum += i % 2 === 1 ? value : 0;
    }

    return fairWaysCount;
};

```

Time and Space Complexity

The time complexity of the provided code is `O(n)`, where `n` is the length of the `nums` list. This is because the code iterates through the list just once with a single loop, and within that loop, the operations performed (including arithmetic operations and conditional checks) are all constant time operations.

The space complexity of the code is `O(1)`. Only a fixed number of variables (`s1`, `s2`, `ans`, `t1`, `t2`) are used, and their space requirement does not scale with the size of the input list `nums`. No additional data structures that grow with the input size are used.