2422. Merge Operations to Turn Array Into a Palindrome

Medium Greedy Array Two Pointers

Leetcode Link

Problem Description

You are given an array called `nums` that contains only positive integers. Your task is to make this array a palindrome. A palindrome is a sequence that reads the same forward and backward. To do this, you are allowed to perform as many operations as you want. An operation consists of selecting any two adjacent elements in `nums` and replacing them with their sum. The goal is to find the minimum number of such operations required to turn the array into a palindrome.

Intuition

The key to solving this problem lies in understanding how a palindrome is structured: the values on the left side of the array must be mirrored on the right side. The strategy is to iteratively make the sums of the values from both ends (left and right) of the array equal, so they form a palindrome.

To implement this strategy, two pointers approach is used, one starting at the left end (beginning of the array) and one at the right end (end of the array). We compare the sums of the elements at these pointers.

- If the left sum is less than the right sum, this means we need to increase the left sum by moving the left pointer to the right and adding the value at the new pointer position to the left sum, counting this action as one operation.
- Conversely, if the right sum is less than the left sum, we move the right pointer to the left, add the value at the new pointer to the right sum, and count it as an operation as well.
- When both sums are equal, we effectively have a palindrome within those boundaries. We move both pointers inward, skipping over the elements we just confirmed as part of the palindrome because they do not need any more operations.
- This process repeats until the pointers meet, which would mean the entire array has become a palindrome.

The trick here is that we don't need to actually replace numbers or keep track of the modified array; instead, we just need to know the count of operations required, which is tallied every time we move either pointer to adjust the sums.

The solution's complexity is O(n), where n is the length of the array, because we possibly go through the array only once with our two pointers.

Solution Approach

The solution uses a two-pointers algorithm to walk through the array from both ends towards the center. This approach helps in reducing the problem to smaller subproblems by considering the current sum at both ends. Here's a step-by-step walkthrough:

- Initialize two pointers, `i` at the start and `j` at the end of the array.
- Initialize two variables, `a` and `b`, to keep track of the sum of the numbers pointed by `i` and `j`. Initially, `a` is assigned `nums[i]`, and `b` is assigned `nums[j]`.
- Initialize a counter `ans` with the value 0 to keep track of the number of operations performed.
- Enter a while loop, which will continue to execute as long as `i < j` (ensuring that we are not comparing the same element with itself or crossing over, which would mean the entire array is a palindrome):
  - Compare the values of `a` and `b`.
  - If `a < b`, we need to increase `a` to eventually match `b`. We do so by incrementing `i` (move the left pointer to the right), adding `nums[i]` to `a`, and incrementing the counter `ans` to represent an operation performed.
  - If `b < a`, similarly, we need to increase `b` to match `a`. Decrement `j` (move the right pointer to the left), add `nums[j]` to `b`, and increment the counter `ans`.
  - If `a == b`, it means that the values from `nums[i]` to `nums[j]` can be part of the palindrome. Therefore, we increment `i`, decrement `j`, and update `a` and `b` with the values at the new indices `nums[i]` and `nums[j]`, respectively. No operation is counted in this case as `a` and `b` are already equal.
- Continue the loop until all elements have been accounted for in pairs that form the palindrome.
- The iterator `ans` is returned as it now contains the minimum number of operations needed to make the array a palindrome.

This simple yet elegant solution leverages the two-pointer technique, which is efficient when you need to compare or pair up elements from opposite ends of an array. It skillfully avoids the need for extra space to store interim arrays, mutating only counters and making the solution very space-efficient (O(1) space complexity).

Example Walkthrough

Let's consider an example to understand the solution approach:

Suppose we are given the following array `nums`:

```
nums = [1, 3, 4, 2, 2]
```

To make `nums` a palindrome using the fewest operations, we will follow the steps outlined:

1. Initialize two pointers, `i` at the start (0) and `j` at the end (4) of the array.
2. Initialize variables `a` with `nums[i]` (which is 1) and `b` with `nums[j]` (which is 2).
3. Initialize the operation counter `ans` to 0.

So, the initial setting looks like:

```
i = 0 - [1, 3, 4, 2, 2] - j = 4
a = 1
b = 2
ans = 0
```

Now, we start iterating:

- Since `a` (1) < `b` (2), we increment `i` to 1 and update `a` by adding `nums[i]`, which makes `a = 1 + 3 = 4` and `ans` becomes 1.
- At this point, `a` (4) == `b` (2), but for the array to be palindrome, `a` and `b` must have the same sum. Thus, we decrease `j` to 3 and update `b` by adding `nums[j]`, now `b = 2 + 2 = 4`, and `ans` becomes 2.
- Now `i` (4) == `j` (4), so we increment `i` to 2, and decrement `j` to 2, effectively skipping over the elements we just confirmed to form the correct structure in our palindrome.

Finally, since `i` now equals `j`, we've considered the entire array, so we finish.

At the end of the process, `ans` equals 2, indicating the minimum number of operations required to turn the array into a palindrome.

So, our example array `nums` can be transformed into a palindrome with a minimum of 2 operations:

- Combine `nums[0]` and `nums[1]` to form `[4, 4, 2, 2]`
- Combine `nums[2]` and `nums[3]` to form `[4, 6, 2]`
- No further operations are needed, as `[4, 6, 2]` is already a palindrome.

Python Solution

```python
from typing import List

class Solution:
    def minimumOperations(self, nums: List[int]) -> int:
        # Initialize two pointers for the start and end of the list
        left_index, right_index = 0, len(nums) - 1

        # Initialize a variable to store the sum of operations performed
        operations_count = 0

        # Loop until the two pointers meet or cross each other
        while left_index < right_index:

            # If the sum on the left is less than the sum on the right,
            # move the left pointer to the right and add the new element to left_sum
            if left_sum < right_sum:
                left_index += 1
                left_sum += nums[left_index]
                operations_count += 1
            # If the sum on the right is less than the sum on the left,
            # move the right pointer to the left and add the new element to right_sum
            elif right_sum < left_sum:
                right_index -= 1
                right_sum += nums[right_index]
                operations_count += 1
            # If the sums are equal, move both pointers and update the sums
            else:
                left_index += 1
                right_index -= 1
                # Check if pointers are still within the array bounds
                if left_index < right_index:
                    left_sum = nums[left_index]
                    right_sum = nums[right_index]

        # Return the total number of operations to make segments equal
        return operations_count
```

Java Solution

```java
class Solution {

    /**
     * Find minimum number of operations to make sum of elements on the left
     * equal to the sum of elements on the right.
     *
     * @param nums Array of integers.
     * @return The minimum number of operations required.
     */
    public int minimumOperations(int[] nums) {
        // Initialize pointers for the left and right parts.
        int leftIndex = 0;
        int rightIndex = nums.length - 1;

        // Initialize sums for the left and right parts.
        long leftSum = nums[leftIndex];
        long rightSum = nums[rightIndex];

        // Variable to keep track of the number of operations performed.
        int operationsCount = 0;

        // Loop until the pointers meet.
        while (leftIndex < rightIndex) {
            // If the left sum is smaller, move the left pointer to the right and add the value to leftSum.
            if (leftSum < rightSum) {
                leftSum += nums[++leftIndex];
                operationsCount++;
            // If the right sum is smaller, move the right pointer to the left and add the value to rightSum.
            } else if (rightSum < leftSum) {
                rightSum += nums[--rightIndex];
                operationsCount++;
            // If both sums are equal, move both pointers and reset the sums.
            } else {
                // Ensure that we do not cross the pointers.
                if (leftIndex + 1 < rightIndex - 1) {
                    leftSum = nums[++leftIndex];
                    rightSum = nums[--rightIndex];
                } else {
                    // Pointers will cross after this step, hence we should break the loop.
                    break;
                }
            }
        }

        // Return the number of operations performed to make the sums equal.
        return operationsCount;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int minimumOperations(vector<int>& nums) {
        // Initialize two pointers from both ends of the array
        int leftIndex = 0, rightIndex = nums.size() - 1;
        // Initialize sum variables for the two pointers
        long leftSum = nums[leftIndex], rightSum = nums[rightIndex];
        // Initialize a variable to count the number of operations performed
        int operationCount = 0;

        // Loop until the two pointers meet
        while (leftIndex < rightIndex) {
            // If leftSum < rightSum, move the left pointer to the right
            if (leftSum < rightSum) {
                // If the left sum is smaller, move the left pointer to the right
                // and add the next number to the left sum.
                leftSum += nums[++leftIndex];
                // Increment the number of operations
                ++operationCount;
            } else if (rightSum < leftSum) {
                // If the right sum is smaller, move the right pointer to the left
                // and add the next number to the right sum.
                rightSum += nums[--rightIndex];
                // Increment the number of operations
                ++operationCount;
            } else {
                // If the sums are equal, move both pointers towards the centre
                // and start the next comparisons with the next outermost numbers
                leftSum = nums[++leftIndex];
                rightSum = nums[--rightIndex];
            }
        }
        // Return the number of operations performed to make sums equal
        return operationCount;
    }
};
```

Typescript Solution

```typescript
/**
 * Calculates the minimum number of operations to make the left sum
 * and right sum equal by only moving elements from one end to the other.
 * @param nums - The array of numbers.
 * @return The number of operations needed to equate the two sums.
 */
function minimumOperations(nums: Array<number>): number {
    // Initialize pointers for both ends of the array
    let leftIndex: number = 0;
    let rightIndex: number = nums.length - 1;

    // Initialize sum variables for the pointers
    let leftSum: number = nums[leftIndex];
    let rightSum: number = nums[rightIndex];

    // Initialize the count of operations
    let operationCount: number = 0;

    // Loop until the pointers meet or cross
    while (leftIndex < rightIndex) {
        if (leftSum < rightSum) {
            // If left sum is smaller, include the next element into the left sum
            leftSum += nums[++leftIndex];
            operationCount++;
        } else if (rightSum < leftSum) {
            // If right sum is smaller, include the next element into the right sum
            rightSum += nums[--rightIndex];
            operationCount++;
        } else {
            // If sums are equal, move to the next elements
            leftSum = nums[++leftIndex];
            rightSum = nums[--rightIndex];
        }
    }

    // Return the total number of operations performed
    return operationCount;
}
```

Time and Space Complexity

Time Complexity

The given code iterates through the `nums` list using two pointers `i` and `j` that start at opposite ends of the list and move towards the center. The main loop runs while `i < j`, so in the worst case, it may iterate through all the elements once. Therefore, the worst-case time complexity is `O(n)`, where `n` is the number of elements in the `nums` list.

In each iteration of the while loop, the code performs constant-time operations—comparisons and basic arithmetic—so these do not affect the overall `O(n)` time complexity.

Space Complexity

Since the algorithm operates in place and the amount of additional memory used does not depend on the input size (`nums` list), the space complexity is `O(1)`. Only a fixed number of variables `i`, `j`, `a`, `b`, and `ans` are used, which occupies constant space irrespective of the input size.