1013. Partition Array Into Three Parts With Equal Sum

Easy Greedy Array

Problem Description

The problem asks us to determine whether it's possible to partition a given array of integers `arr` into three contiguous, non-empty parts such that the sum of the numbers in each part is equal. This is akin to cutting the array at two points to create three subarrays, where each subarray sums up to the same value. The constraints are quite simple: we need to find two indices `i` and `j`, with `i + 1 < j`, that satisfy the partitioning condition.

Intuition

To tackle this problem, we need to consider that if we can partition the array into three parts with equal sums, the sum of the whole array must be divisible by 3. That's our starting point. If the entire sum `s` does not satisfy this condition, we can immediately return `False`, because no such partitioning will be possible.

If, on the other hand, `s` is divisible by 3, we then look for the two cut points by trying to find a subarray from the start and from the end that each sum up to `s // 3` (which represents one-third of the array's total sum).

We iterate from the start of the array, accumulating a sum `a` until `a` equals `s // 3`. This represents the end of the first part of the array. We do a similar process from the end of the array, accumulating another sum `b` until `b` equals `s // 3`. This loop runs in reverse and represents the start of the third part of the array. If we find such indices `i` and `j` where `a` and `b` each equal `s // 3`, and `i` is strictly less than `j - 1` (which ensures that the middle part is non-empty), we can conclude that it is indeed possible to partition the array into three parts with equal sums, and return `True`. Otherwise, the function returns `False`.

Solution Approach

To implement the solution, the algorithm takes the following steps, leveraging simple iteration and summation:

1. Initially, the algorithm calculates the sum `s` of all elements in the array. Since we aim to partition the array into three parts with equal sum, we first check if the total sum is divisible by 3. If not, we return `False`.
2. To find the partitioning of the array, we maintain two pointers `i` and `j` representing the boundaries of the first and last section of the partitioned array, respectively. We also maintain two accumulator variables `a` and `b` for the sums of the respective sections.
3. We start a while loop that runs as long as `i` has not reached the end of the array. Within this loop, we increment `a` with the current element `arr[i]` and check if `a` has reached `s // 3`. If it has, we break the loop, as we have found the sum for the first part.
4. Similarly, we start a while loop that runs in reverse as long as `j` has not reached the start of the array (`~j` checks for this since `~j` will be `-1` when `j` is `0`). We increment `b` with the current element `arr[j]` and check if `b` has reached `s // 3`. If it has, we break the loop.
5. After finding the two potential cut points `i` and `j`, we check if `i < j - 1`. This ensures that there is at least one element between the two parts that we have found, fulfilling the non-empty middle section requirement. If this condition is true, we return `True`, indicating that the array can be partitioned into three parts with equal sums.

The algorithm mainly uses two pointers with a linear pass from both ends of the array, making the time complexity O(n), where n is the number of elements in the array. The space complexity is O(1) since it only uses a fixed number of extra variables.

Example Walkthrough

Let's say we have the following array of integers:

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

Now, let's walk through the solution approach step by step with this example:

1. First, we calculate the sum `s` of all elements in the array `arr`. We get `s = 1 + 3 + 4 + 2 + 2 = 12`. Since 12 is divisible by 3, we can proceed. If the sum were not divisible by 3, we would return `False` as it would be impossible to partition the array into three parts with an equal sum.
2. We then initialize two pointers `i` with the value 0 and `j` with the value `len(arr) - 1`. Also, we initialize two accumulator variables `a` and `b` to 0, which we will use to store the sums of the parts as we loop through `arr`.
3. The target sum for each part is `s // 3`, which is `12 // 3 = 4`. We start looping through the array from the start, adding each element to `a` until `a` equals 4. Adding the first element 1, `a` becomes 1. Adding the next element 3, `a` becomes 4. We have found the end of the first part at `i = 1`.
4. Similarly, we loop through the array from the end, decrementing `j` and adding each element to `b`. Adding the last element 2, `b` becomes 2. Decrementing `j`, we add the next last element 2, `b` becomes 4. We have found the beginning of the third part at `j = 3`.
5. We then check if `i < j - 1`. In this case, `i = 1`, and `j = 3`, so `1 < 3 - 1` which is `1 < 2`. This inequality holds, meaning there is at least one element for the middle part, and we can confirm that the array can be split into three parts with equal sums `[1, 3]`, `[4]`, and `[2, 2]`.

Since all checks pass, the function would return `True` for this array, indicating that partitioning into three parts with equal sums is indeed possible.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def can_three_parts_equal_sum(self, arr: List[int]) -> bool:
        # Calculate the total sum of all elements in the array
        total_sum = sum(arr)
        # If the total sum is not divisible by 3, it's not possible to partition the array into three parts with equal sums
        if total_sum % 3 != 0:
            return False

        # Initialize pointers for the left and right partitions
        left_index, right_index = 0, len(arr) - 1
        left_sum, right_sum = 0, 0

        # Increase the left partition until we find a partition with a sum that's a third of the total sum
        while left_index < len(arr):
            left_sum += arr[left_index]
            if left_sum == total_sum // 3:
                break
            left_index += 1

        # Similarly, decrease the right partition to find a partition with a sum that's a third of the total sum
        while right_index >= 0:
            right_sum += arr[right_index]
            if right_sum == total_sum // 3:
                break
            right_index -= 1

        # Check if there is at least one element between the two partitions
        # i + 1 < j ensures there's room for a middle partition
        return left_index < right_index - 1

# Example usage:
# solution = Solution()
# arr = [0, 2, 1, -6, 6, -7, 9, 1, 2, 0, 1]
# print(solution.can_three_parts_equal_sum(arr))  # Output: True
```

Java

```java
class Solution {
    public boolean canThreePartsEqualSum(int[] arr) {
        // Calculate the total sum of the elements in the array
        int totalSum = 0;
        for (int num : arr) {
            totalSum += num;
        }

        // If the total sum is not divisible by 3, we cannot split the array into 3 parts with equal sum
        if (totalSum % 3 != 0) {
            return false;
        }

        // Initialize two pointers for traversing the array from both ends
        int start = 0, end = arr.length - 1;

        // Initialize sums for the segments from the start and end of the array
        int startSum = 0, endSum = 0;

        // Find the first segment from the start that sums up to a third of the total sum
        while (start < arr.length) {
            startSum += arr[start];
            if (startSum == totalSum / 3) {
                break;
            }
            ++start;
        }

        // Find the first segment from the end that sums up to a third of the total sum
        while (end >= 0) {
            endSum += arr[end];
            if (endSum == totalSum / 3) {
                break;
            }
            --end;
        }

        // Check if there is a middle segment between the two previously found segments
        // The condition `start < end - 1` ensures there is at least one element in the middle segment
        return start < end - 1;
    }
}
```

C++

```cpp
#include <vector>
using namespace std;

class Solution {
public:
    // This function checks whether we can partition the array into three parts with equal sum
    bool canThreePartsEqualSum(vector<int>& arr) {
        int totalSum = 0;

        // Calculate the sum of all elements in the array
        for (int value : arr) {
            totalSum += value;
        }

        // If the total sum is not divisible by 3, we can't partition it into three parts with equal sum
        if (totalSum % 3 != 0) return false;

        // The target sum for each of the three parts
        int targetSum = totalSum / 3;

        // Indices to iterate over the array from start (i) and end (j)
        int i = 0, j = arr.size() - 1;
        int sumFromStart = 0, sumFromEnd = 0;

        // Find the partition from the start of the array
        while (i < arr.size()) {
            sumFromStart += arr[i];
            // If we've reached the target sum, stop the loop
            if (sumFromStart == targetSum) {
                break;
            }
            ++i;
        }

        // Find the partition from the end of the array
        while (j >= 0) {
            sumFromEnd += arr[j];
            // If we've reached the target sum, stop the loop
            if (sumFromEnd == targetSum) {
                break;
            }
            --j;
        }

        // After finding the two partitions, check if there is at least one element between them
        return i < j - 1;
    }
};

// Example usage:
// int main() {
//     Solution obj;
//     vector<int> arr = {0, 2, 1, -6, 6, -7, 9, 1, 2, 0, 1};
//     bool result = obj.canThreePartsEqualSum(arr);
//     // Expected output: true, because the array can be partitioned as [0, 2, 1], [-6, 6, -7, 9, 1], [2, 0, 1] with equal sum
// }
```

TypeScript

```typescript
function canThreePartsEqualSum(arr: number[]): boolean {
    let totalSum: number = 0;

    // Calculate the sum of all elements in the array
    for (let value of arr) {
        totalSum += value;
    }

    // If the total sum is not divisible by 3, we can't partition it into three parts with equal sum
    if (totalSum % 3 !== 0) return false;

    // The target sum for each of the three parts
    let targetSum: number = totalSum / 3;

    // Indices to iterate over the array from start (i) and end (j)
    let i: number = 0, j: number = arr.length - 1;
    let sumFromStart: number = 0, sumFromEnd: number = 0;

    // Find the partition from the start of the array
    while (i < arr.length) {
        sumFromStart += arr[i];
        // If we've reached the target sum, stop the loop
        if (sumFromStart === targetSum) {
            break;
        }
        ++i;
    }

    // Find the partition from the end of the array
    while (j >= 0) {
        sumFromEnd += arr[j];
        // If we've reached the target sum, stop the loop
        if (sumFromEnd === targetSum) {
            break;
        }
        --j;
    }

    // After finding the two partitions, check if there is at least one element between them
    return i < j - 1;
}

// Example usage:
// const arr: number[] = [0, 2, 1, -6, 6, -7, 9, 1, 2, 0, 1];
// const result: boolean = canThreePartsEqualSum(arr);
// This would return true, because the array can be partitioned as [0, 2, 1], [-6, 6, -7, 9, 1], [2, 0, 1] with each part summing
```

```python
from typing import List

class Solution:
    def can_three_parts_equal_sum(self, arr: List[int]) -> bool:
        # Calculate the total sum of all elements in the array
        total_sum = sum(arr)
        # If the total sum is not divisible by 3, it's not possible to partition the array into three parts with equal sums
        if total_sum % 3 != 0:
            return False

        # Initialize pointers for the left and right partitions
        left_index, right_index = 0, len(arr) - 1
        left_sum, right_sum = 0, 0

        # Increase the left partition until we find a partition with a sum that's a third of the total sum
        while left_index < len(arr):
            left_sum += arr[left_index]
            if left_sum == total_sum // 3:
                break
            left_index += 1

        # Similarly, decrease the right partition to find a partition with a sum that's a third of the total sum
        while right_index >= 0:
            right_sum += arr[right_index]
            if right_sum == total_sum // 3:
                break
            right_index -= 1

        # Check if there is at least one element between the two partitions
        # i + 1 < j ensures there's room for a middle partition
        return left_index < right_index - 1

# Example usage:
# solution = Solution()
# arr = [0, 2, 1, -6, 6, -7, 9, 1, 2, 0, 1]
# print(solution.can_three_parts_equal_sum(arr))  # Output: True
```

Time and Space Complexity

Time Complexity

The time complexity of the provided code can be analyzed through its operations step by step.

1. Calculation of the sum - This involves summing all the elements in the array once, which takes `O(N)` time, where `N` is the length of the array.
2. Two separate while loops to find the partition points:

- The first while loop iterates at most `N` times to find the point where the first third of the array sum can be found (`a == s // 3`). In the worst case, this is `O(N)`.
- The second while loop is similar, working backwards from the end of the array. This also has a worst-case scenario of `O(N)`.

Since these loops do not overlap and are not nested, the overall time complexity is the sum of these individual components, which would still be `O(N)`.

Space Complexity

The space complexity of the algorithm is quite straightforward:

- No additional data structures that grow with the input size are introduced.
- Only a fixed amount of extra variables (`s`, `i`, `j`, `a`, `b`) are used regardless of the input size.

Thus, the space complexity of the function is `O(1)`, indicating constant space usage.