2025. Maximum Number of Ways to Partition an Array

Hard   Array   Hash Table   Counting   Enumeration   Prefix Sum   Leetcode Link

Problem Description

You are given an array `nums` with `n` integers. We want to determine the maximum number of ways we can partition this array into two non-empty parts by choosing a `pivot` index so that the sum of elements to the left of the pivot is equal to the sum of elements to the right of the pivot. You can also change one element in the array to another integer `k` to increase the number of such partitions.

Partitioning an array means picking an index `pivot` such that `1 <= pivot < n` and ensuring the following condition is met: the sum of elements from `nums[0]` to `nums[pivot - 1]` should be equal to the sum of elements from `nums[pivot]` to `nums[n - 1]`.

You are permitted to change at most one element in `nums` to `k` or you can choose to keep the array as it is. The goal is to find the maximum number of partitions that satisfy the condition described above after making at most one such change.

Intuition

To arrive at the solution, we need to think about how changing one element to `k` could affect the number of valid partitions. When we don't change any element, any `pivot` that splits the array into two equal sums is a valid partition. We can calculate the total sum of the array first and then iterate through the array while keeping track of the cumulative sum. This helps us to determine if a `pivot` is a valid partition for the unchanged array.

However, when considering the change of one element to `k`, we need to think about where this operation would be most beneficial. Changing an element affects the sums of the elements to the left and to the right of that element. It's helpful to keep track of cumulative sums from the left and the right, and update these sums as if the change to `k` has been made.

We use two dictionaries: `left` to count the frequency of the left sums and `right` for the right sums. For each element `x` in the array, if we replace `x` with `k`, we calculate the difference `d = k - x`. We then update our counters based on this difference to reflect the new sums if the element were replaced.

Finally, we iterate over each element, consider replacing it with `k`, and count how many times the left and right sums across the pivot will then be equal. We update the answer with this count whenever it results in more valid partitions. The idea is to maximize this count, which gives us the maximum possible number of ways to partition the array after changing at most one element.

Solution Approach

The solution leverages the `defaultdict` from the `collections` module, a subclass of the built-in `dict` class. It's used to store key-value pairs where the keys represent sums of elements and the values count how many times each sum occurs.

Here's a step-by-step explanation of the implementation:

1. Initialize the cumulative sum list `s` containing the same value `nums[0]`. This list will help us keep track of the sum from the beginning of the array up to each index.
2. Create two dictionaries, `right` and `left`, using the `defaultdict` with integer default types. The `right` dictionary will keep track of sums from the right side of the array, and `left` will do so for the left side.
3. Iterate through the array, starting from index 1, to create the cumulative sum array `s` and populate the `right` dictionary where each sum prior to the current index will be used as a key, and its frequency as the value.
4. Set `ans` to 0. This variable will hold the count of the maximum number of ways to partition the array.
5. Check if the total sum `s[-1]` is even. If yes, set `ans` to the number of ways to partition the array without changing an element. This can be found directly from the `right` dictionary for half of the total sum, since that would mean the array can be bisected into two equal halves.
6. Iterate over each element and the values from the cumulative sum list together. In each iteration, calculate the difference `d = k - x`, which represents how the sum would change if we replaced an element with `k`.
7. For each value `v` in the cumulative sum list and each `x` in the `nums`, check if the total sum of the array after the change, which is `s[-1] + d`, is even. If it is, compute the count `t` as the sum of the values from the `left` and `right` dictionaries at the respective adjusted sums, which would represent a valid midpoint after the change. Compare this with our current `ans` and update `ans` if `t` is larger.
8. Increment the count of the current sum in the `left` dictionary by 1, and decrement the count of the current sum in the `right` dictionary by 1, reflecting that we are moving from right to left in terms of possible pivots.
9. Finally, return the value stored in `ans`, which represents the maximum ways we can partition the array after making at most one change.

The algorithm efficiently counts the number of ways to partition the array by maintaining cumulative sums and dynamically adjusting the counts when considering the impact of one element change.

Example Walkthrough

Let's consider an example array `nums` as `[1, 2, 3, 4, 6]` and suppose we want to change one element to `k = 5`. We want to calculate the maximum number of ways we can partition the array such that the sum of elements on either side of the pivot is equal.

1. Initialize the cumulative sum list `s`. After iterating over `nums`, `s` will be `[1, 3, 6, 10, 16]`.
2. Create two dictionaries, `right` and `left`, using the `defaultdict(int)`.
3. Populate the `right` dictionary, recording the frequence of sums to the right side of the array so that it looks like this: `{16: 1, 15: 1, 13: 1, 10: 1, 6: 1}` before any changes, signifying that the sum 16 occurs once, 15 occurs once, and so on.
4. Set `ans` to 0, ready to hold the maximum number of partitions.
5. The total sum `s[-1]` is 16, which is not even, so we cannot initially partition the array without changing an element.
6. Now, iterate over `nums` and `s`:
   - At index 0, `x = 1`, the difference `d = k - x = 5 - 1 = 4`. The total sum after the change would be `16 + 4 = 20`, which is even.
   - At index 1, `x = 2`, `d = 5 - 2 = 3`. The total sum would be `16 + 3 = 19`, which is not even, so no partition is possible here.
   - Continuing this process, we find that for index 2, `x = 3`, `d = 5 - 3 = 2`, and the total sum is even at `18`, suggesting a potential partition.
7. When `x = 3` and `v = 6` at index 2, check for potential partitions:
   - The left sum would be `6 + 2 = 8`, and the right sum (excluding the current element) would also need to be 8 for a valid partition.
   - We add `1` to `left[8]` as this sum has now occurred once.
   - We decrease `right[10]` to `0` since we have used that partition possibility.
   - We find that the partition could occur at index 3, since the sums on either side would then be equal.
8. Increment the left dictionary and decrement the right dictionary as we move along the array and continue checking for partitions after each element.
9. Return the value stored in `ans`, which is the maximum ways we can partition the array. In this case, `ans` would be updated to `1` as we found a valid partition when we considered changing the third element to `5`, which would make the sums on either side of the pivot equal.

This walkthrough demonstrates how the algorithm works in a practical example, changing one element to `k` and finding the maximum number of partitions where the sums on either side of the pivot are equal.

Python Solution

```python
from collections import defaultdict

class Solution:
    def waysToPartition(self, nums: List[int], k: int) -> int:
        # Determine the number of elements in the list.
        num_elements = len(nums)

        # Initialize the prefix sum array.
        prefix_sums = [nums[0]] * num_elements

        # Create a defaultdict for counting right partitions.
        right_partitions_counts = defaultdict(int)

        # Calculate the prefix sums and populate the right partitions.
        for i in range(1, num_elements):
            prefix_sums[i] = prefix_sums[i - 1] + nums[i]
            right_partitions_counts[prefix_sums[i - 1]] += 1

        # Initialize the answer to 0.
        max_possible_ways = 0

        # Check if the total sum is even and set the initial maximum possible ways if true.
        if prefix_sums[-1] % 2 == 0:
            max_possible_ways = right_partitions_counts[prefix_sums[-1] // 2]

        # Create a defaultdict for counting left partitions.
        left_partitions_counts = defaultdict(int)

        # Iterate over the list to find the number of ways we can partition.
        for prefix_sum, num in zip(prefix_sums, nums):
            diff = k - num

            # Check if the adjusted total sum is even and update the maximum possible ways.
            if (prefix_sums[-1] + diff) % 2 == 0:
                total_ways = left_partitions_counts[(prefix_sums[-1] + diff) // 2] \
                             + right_partitions_counts[(prefix_sums[-1] - diff) // 2]
                max_possible_ways = max(max_possible_ways, total_ways)

            # Update the left and right partitions counts.
            left_partitions_counts[prefix_sum] += 1
            right_partitions_counts[prefix_sum] -= 1

        # Return the maximum number of ways to partition the list.
        return max_possible_ways

```

Java Solution

```java
import java.util.HashMap;
import java.util.Map;

class Solution {
    public int waysToPartition(int[] nums, int k) {
        // Initialize the length of the input array.
        int length = nums.length;
        // Prefix sum array to keep track of the sums.
        int[] prefixSums = new int[length];
        // Initializing the prefix sum array with the first element.
        prefixSums[0] = nums[0];
        // A map to store count of prefix sums to the right of a partition.
        Map<Integer, Integer> rightPartitions = new HashMap<>();
        // Calculate the prefix sums and populate the right partitions map.
        for (int i = 0; i < length - 1; ++i) {
            rightPartitions.merge(prefixSums[i], 1, Integer::sum);
            prefixSums[i + 1] = prefixSums[i] + nums[i + 1];
        }
        // Initialize the max number of ways to partition.
        int maxWays = 0;
        // Check if the total sum is even. If yes, there could exist a partition.
        if (prefixSums[length - 1] % 2 == 0) {
            maxWays = rightPartitions.getOrDefault(prefixSums[length - 1] / 2, 0);
        }
        // A map to store count of prefix sums to the left of a partition.
        Map<Integer, Integer> leftPartitions = new HashMap<>();
        // Evaluating each possible partition.
        for (int i = 0; i < length; ++i) {
            // Calculate the difference between the target and the current element.
            int difference = k - nums[i];
            // Check if after replacing nums[i] with k makes the sum even.
            if ((prefixSums[length - 1] + difference) % 2 == 0) {
                int t = leftPartitions.getOrDefault((prefixSums[length - 1] + difference) / 2, 0)
                    + rightPartitions.getOrDefault((prefixSums[length - 1] - difference) / 2, 0);
                // Update the max ways to the larger of the current max ways and the temporary count.
                maxWays = Math.max(maxWays, t);
            }
            // Update the maps for the next iteration.
            leftPartitions.merge(prefixSums[i], 1, Integer::sum);
            rightPartitions.merge(prefixSums[i], -1, Integer::sum);
        }
        // Return the maximum number of ways we can partition.
        return maxWays;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int waysToPartition(vector<int>& nums, int k) {
        int totalNumbers = nums.size(); // Renamed for clarity.

        // This array will hold the cumulative sum from index 0 to 'i'.
        vector<long long> cumulativeSum(totalNumbers, 0);
        cumulativeSum[0] = nums[0];

        // Maps to keep track of number of ways to partition on the right, and on the left.
        unordered_map<long long, int> rightPartitions;

        // Initialize the right partitions map and populate cumulative sum array.
        for (int i = 0; i < totalNumbers - 1; ++i) {
            rightPartitions[cumulativeSum[i]]++;
            cumulativeSum[i + 1] = cumulativeSum[i] + nums[i + 1];
        }

        // Will hold the maximum number of ways to partition array.
        int maxWays = 0;

        // Check if the total sum is even and update ways accordingly.
        if (cumulativeSum[totalNumbers - 1] % 2 == 0) {
            maxWays = rightPartitions[cumulativeSum[totalNumbers - 1] / 2];
        }

        unordered_map<long long, int> leftPartitions; // For partitions on the left.

        // Iterate over each number to consider it as the partition point.
        for (int i = 0; i < totalNumbers; ++i) {
            // Check if by replacing nums[i] with k, whether we can increase the number of ways.
            int difference = k - nums[i];
            if ((cumulativeSum[totalNumbers - 1] + difference) % 2 == 0) {
                int currentWays =
                    leftPartitions[(cumulativeSum[totalNumbers - 1] + difference) / 2] +
                    rightPartitions[(cumulativeSum[totalNumbers - 1] - difference) / 2];
                maxWays = max(maxWays, currentWays);
            }

            // Update the maps to reflect the change in partition position.
            leftPartitions[cumulativeSum[i]]++;
            rightPartitions[cumulativeSum[i]]--;
        }
        return maxWays; // Return the maximum number of ways found.
    }
};

```

Typescript Solution

```typescript
// Function to calculate the number of ways to partition a given array
function waysToPartition(nums: number[], k: number): number {
    let totalNumbers: number = nums.length; // Renaming for clarity

    // This array will hold the cumulative sum from index 0 to 'i'.
    let cumulativeSum: number[] = new Array(totalNumbers).fill(0);
    cumulativeSum[0] = nums[0];

    // Maps to keep track of number of ways to partition on the right, and on the left.
    let rightPartitions: Map<number, number> = new Map<number, number>();

    // Initialize the right partitions map and populate cumulative sum array.
    for (let i: number = 0; i < totalNumbers - 1; ++i) {
        rightPartitions.set(cumulativeSum[i], (rightPartitions.get(cumulativeSum[i]) || 0) + 1);
        cumulativeSum[i + 1] = cumulativeSum[i] + nums[i + 1];
    }

    // Will hold the maximum number of ways to partition array.
    let maxWays: number = 0;

    // Check if the total sum is even and update ways accordingly.
    if (cumulativeSum[totalNumbers - 1] % 2 === 0) {
        maxWays = rightPartitions.get(cumulativeSum[totalNumbers - 1] / 2) || 0;
    }

    let leftPartitions: Map<number, number> = new Map<number, number>(); // For partitions on the left.

    // Iterate over each number to consider it as the partition point.
    for (let i: number = 0; i < totalNumbers; ++i) {
        // Check if by replacing nums[i] with k, whether we can increase the number of ways.
        let difference: number = k - nums[i];
        if ((cumulativeSum[totalNumbers - 1] + difference) % 2 === 0) {
            let currentWays: number =
                (leftPartitions.get((cumulativeSum[totalNumbers - 1] + difference) / 2) || 0) +
                (rightPartitions.get((cumulativeSum[totalNumbers - 1] - difference) / 2) || 0);
            maxWays = Math.max(maxWays, currentWays);
        }

        // Update the maps to reflect the change in partition position.
        leftPartitions.set(cumulativeSum[i], (leftPartitions.get(cumulativeSum[i]) || 0) + 1);
        rightPartitions.set(cumulativeSum[i], (rightPartitions.get(cumulativeSum[i]) || 1) - 1);
    }
    return maxWays; // Return the maximum number of ways found.
}

```

Time and Space Complexity

Time Complexity

The given Python function `waysToPartition` performs a series of operations on an array to determine the number of ways it can be partitioned such that the sums of the elements on either side of the partition are equal, with the option of changing one element to `k`.

First, the function creates an accumulated sum array `s` of the input array `nums`. This operation traverses the array once and has a time complexity of `O(n)`, where `n` is the length of the input array.

Next, the function initializes a `right` dictionary (from the `collections.defaultdict` class) to count the occurrences of each prefix sum as we iterate from the end to the beginning of the accumulated sum array. Filling this dictionary also takes `O(n)`.

The function then checks a special case where the total sum of the array is even, and a partition is possible without any change. Checking and updating the count takes constant time `O(1)`.

Following this, the function iterates through `nums` and its accumulated sum `s` simultaneously and updates two dictionaries: `left` and `right`. Each iteration involves a constant amount of work - updating dictionaries and a few arithmetic operations. Since the iteration happens `n` times, we get `O(n)`.

Within the same loop, the partition count is updated based on whether it's possible to create an equal partition by changing the current number to `k`. The checks and updates are performed in constant time for each iteration.

Considering all these operations together, since they are sequential, the total time complexity is **`O(n)` + `O(n)` + `O(1)` + `O(n)` = `O(n)`**.

Space Complexity

The space complexity of the function involves the space taken by the input `nums`, the accumulated sum array `s`, and the `left` and `right` dictionaries.

The accumulated sum array `s` has the same length as `nums`, contributing `O(n)` space complexity.

Both `left` and `right` dictionaries could, in the worst case, store a distinct sum for every prefix and suffix, which also contributes up to `O(n)` space complexity.

Therefore, the total space complexity is the sum of the space complexities of these data structures, **`O(n)` + `O(n)` + `O(n)` = `O(n)`**.