416. Partition Equal Subset Sum

Medium Array Dynamic Programming

Problem Description

The given problem asks us to determine if we can split an array of integers, `nums`, into two subsets such that the sum of the elements in both subsets is the same. This is essentially asking if `nums` can be partitioned into two subsets of equal sum. If such a partition is possible, we should return `true`, otherwise, we return `false`.

To understand this problem better, imagine you have a set of blocks with different weights, and you want to see if you can divide them into two groups that weigh the same. If it can be done, then each group represents a subset with an equal sum.

Intuition

The solution to this problem is based on the concept of dynamic programming, particularly the 0/1 Knapsack problem, where we aim to find a subset of numbers that can sum up to a specific target, in this case, half the sum of all elements in `nums`.

The intuition behind this solution is:

1. First, we calculate the sum of all elements in the array. If the sum is an odd number, it's impossible to partition the array into two subsets with an equal sum, so we immediately return `false`.
2. If the sum is even, our target becomes half of the total sum, and we set up an array `f` of boolean values that represents if this sum can be reached using a combination of the numbers we've seen so far. `f` is initialized with a size equal to the target plus one (`m + 1`), with the first value `True` (since we can always reach a sum of 0) and the rest `False`.
3. We iterate over each number in our array `nums`. For each number, we update our `f` array from right to left, starting at our target `m` and going down to the value of the number `x`. We do this backward to ensure that each number is only considered once. At each position `j`, we update `f[j]` by checking if `f[j]` was previously true or if `f[j−x]` was true. The latter means that if we already could sum up to `j−x`, then by adding `x`, we can now also sum up to `j`.
4. At the end of this process, `f[m]` tells us whether we've found a subset of elements that sum up to `m`, which would be half the sum of the entire array. If `f[m]` is true, we have our partition and return `true`, otherwise, we return `false`.

Solution Approach

The solution implements a classic dynamic programming approach to solve the subset sum problem, which is a variation of the 0/1 Knapsack problem. Here's a step-by-step guide to understanding the algorithm:

1. **Calculate the Sum and Determine Feasibility:** We begin by finding the sum of all elements in the array using `sum(nums)`. We divide this sum by 2 using the `divmod` function, which gives us the quotient `m` and the remainder `mod`. If `mod` is not zero, the sum is odd, and we cannot partition an odd sum into two equal even halves, so we return `false`.
2. **Dynamic Programming Array Setup:** Next, we set up an array `f` with `m + 1` boolean elements, which will help us track which sums can be achieved from subsets of the array. We initialize `f[0]` to `True` because a zero sum is always possible (the empty subset), and the rest to `False`.
3. **Iterate and Update the DP Array:** For each number `x` in `nums`, we iterate over the array `f` from `m` down to `x`. We do this in reverse order to ensure that each element contributes only once to each sum.
4. **Update the DP Array:** For each position `j` in `f`, we check if `f[j]` was already `True` (sum `j` was already achievable) or if `f[j − x]` was `True`. If `f[j − x]` was `True`, it means there was a subset of previous elements that added up to `j − x`. By including the current element `x`, we can now reach the sum `j`, so we set `f[j]` to `True`.
5. **Return the Result:** Finally, we return the value of `f[m]`. This value tells us whether there is a subset of elements from `nums` that adds up to `m`, which would be half of the total sum. If `f[m]` is `True`, it means we can partition the array into two subsets with an equal sum, and we return `true`; otherwise, we return `false`.

The pattern used in this algorithm leverages the properties of boolean arithmetic wherein `True` represents 1 and `False` represents 0. The statement `f[j] = f[j] or f[j − x]` is an efficient way to update our boolean array because it captures both conditions for setting `f[j]` to `True`: either it's already `True`, or `f[j − x]` is `True` and we just add `x` to reach the required sum `j`.

By re-using the array `f` in each iteration and only considering each number once, we keep our space complexity to O(sum/2), which is much more efficient than trying to store all possible subset sums up to the total sum of the array.

Example Walkthrough

Let's walk through an example to illustrate the solution approach. Consider an array `nums` with the following elements: `[1, 5, 11, 5]`.

1. **Calculate the Sum and Determine Feasibility:**
   - Compute the sum of the elements: `1 + 5 + 11 + 5 = 22`.
   - Use `divmod` to check if the sum is even or odd: `divmod(22, 2)` gives us `(11, 0)`.
   - Since the remainder is `0`, the sum is even, and proceeding is feasible.
2. **Dynamic Programming Array Setup:**
   - Our target sum `m` is `22 / 2 = 11`.
   - Initialize `f` with dimensions `[12]` (`m + 1`) and set `f[0]` to `True`.
3. **Iterate and Update the DP Array:**
   - Start iterating over the array `nums`: `[1, 5, 11, 5]`.
4. **Update the DP Array:**
   - For `x` = 1 (first element), update `f` from `11` down to `1`. Since `f[0]` is `True`, set `f[1]` to `True`.
   - For `x` = 5 (second element), update `f` from `11` down to `5`. Now `f[5]`, `f[6]`, `f[7]`, `f[8]`, `f[9]`, and `f[11]` become `True`.
   - For `x` = 11 (third element), since `f[0]` is `True`, set `f[11]` to `True`. However, `f[11]` is already `True` from the previous step.
   - Lastly, for `x` = 5 (fourth element), update `f` again similarly to when `x` was `5` before.
5. **Return the Result:**
   - After the final iteration, we check the value of `f[11]`, which is `True`.
   - This indicates that there is a subset with a sum of `11`, which is half of the total sum.

Therefore, the array `[1, 5, 11, 5]` can be partitioned into two subsets with equal sum, and we return `true`.

Solution Implementation

Python

```python
class Solution:
    def can_partition(self, nums: List[int]) -> bool:
        # Compute the total sum of the nums array and divide by 2 (partition sum)
        total_sum, remainder = divmod(sum(nums), 2)

        # If the sum of nums is odd, we cannot partition it into two equal subsets
        if remainder:
            return False

        # Initialize a boolean array that will keep track of possible sums
        can_partition = [True] + [False] * total_sum

        # Loop through each number in the nums array
        for num in nums:
            # Check each possible sum in reverse (to avoid using the same number twice)
            for j in range(total_sum, num - 1, -1):
                # Update the can_partition array
                # True if the number itself can form the sum
                # or if the sum can be formed by adding the number to a previously possible sum
                can_partition[j] = can_partition[j] or can_partition[j - num]

        # The last element in the can_partition array indicates if we can partition
        # nums into two equal subsets
        return can_partition[total_sum]
```

Java

```java
class Solution {
    public boolean canPartition(int[] nums) {
        // Calculate the sum of all array elements
        int sum = 0;
        for (int num : nums) {
            sum += num;
        }

        // If the sum is odd, it's not possible to partition the array into two subsets of equal sum
        if (sum % 2 != 0) {
            return false;
        }

        // Target sum for each subset is half of the total sum
        int targetSum = sum / 2;

        // Create a boolean array to store the subset sums achievable up to the targetSum
        boolean[] subsetSums = new boolean[targetSum + 1];

        // There's always one subset with sum 0, the empty set
        subsetSums[0] = true;

        // Check each number in the given array
        for (int num : nums) {
            // Traverse the subsetSums array in reverse to avoid using an element multiple times
            for (int j = targetSum; j >= num; j--) {
                // Update the subset sums that are achievable
                // If j-num is achievable, set j as achievable (because we're adding num to the subset)
                subsetSums[j] = subsetSums[j] || subsetSums[j - num];
            }
        }

        // The result is whether the targetSum is achievable as a subset sum
        return subsetSums[targetSum];
    }
}
```

C++

```cpp
#include <numeric>
#include <vector>
#include <cstring>

class Solution {
public:
    // Function to determine if the input array can be partitioned into two subsets of equal sum
    bool canPartition(vector<int>& nums) {
        // Calculate the sum of elements in the nums array
        int totalSum = accumulate(nums.begin(), nums.end(), 0);

        // If the total sum is odd, it's not possible to divide it into two equal parts
        if (totalSum % 2 == 1) {
            return false;
        }

        // Target sum for each partition
        int targetSum = totalSum >> 1;

        // Create a dynamic programming array to keep track of possible sums
        bool dp[targetSum + 1];

        // Initialize the dynamic programming array to false
        memset(dp, false, sizeof(dp));

        // The sum of 0 is always achievable (by selecting no elements)
        dp[0] = true;

        // Iterate through the numbers in the array
        for (int num : nums) {
            // Check each possible sum in reverse to avoid using a number twice
            for (int j = targetSum; j >= num; --j) {
                // Update the dp array: dp[j] will be true if dp[j - num] was true
                // This means that current number 'num' can add up to 'j' using the previous numbers
                dp[j] = dp[j] || dp[j - num];
            }
        }

        // The result is whether it's possible to achieve the targetSum using the array elements
        return dp[targetSum];
    }
};
```

TypeScript

```typescript
function canPartition(nums: number[]): boolean {
    // Calculate the sum of all elements in the array
    const totalSum = nums.reduce((accumulator, currentValue) => accumulator + currentValue, 0);

    // If the total sum is odd, it's not possible to partition the array into two subsets with an equal sum
    if (totalSum % 2 !== 0) {
        return false;
    }

    // Target sum is half of the total sum
    const targetSum = totalSum >> 1;

    // Initialize a boolean array to keep track of possible subset sums
    const possibleSums = Array(targetSum + 1).fill(false);
    // Always possible to pick a subset with sum 0 (empty subset)
    possibleSums[0] = true;

    // Iterate through all numbers in the given array
    for (const num of nums) {
        // Iterate backwards through possibleSums array to check if current number can contribute to the targetSum
        for (let j = targetSum; j >= num; --j) {
            // Update possibleSums array to reflect the new subset sum that can be formed
            possibleSums[j] = possibleSums[j] || possibleSums[j - num];
        }
    }

    // Return whether a subset with the targetSum is possible
    return possibleSums[targetSum];
}
```


```python
class Solution:
    def can_partition(self, nums: List[int]) -> bool:
        # Compute the total sum of the nums array and divide by 2 (partition sum)
        total_sum, remainder = divmod(sum(nums), 2)

        # If the sum of nums is odd, we cannot partition it into two equal subsets
        if remainder:
            return False

        # Initialize a boolean array that will keep track of possible sums
        can_partition = [True] + [False] * total_sum

        # Loop through each number in the nums array
        for num in nums:
            # Check each possible sum in reverse (to avoid using the same number twice)
            for j in range(total_sum, num - 1, -1):
                # Update the can_partition array
                # True if the number itself can form the sum
                # or if the sum can be formed by adding the number to a previously possible sum
                can_partition[j] = can_partition[j] or can_partition[j - num]

        # The last element in the can_partition array indicates if we can partition
        # nums into two equal subsets
        return can_partition[total_sum]
```


Time and Space Complexity

The code is designed to solve the Partition Equal Subset Sum problem which is to determine if the given set of numbers can be partitioned into two subsets such that the sum of elements in both subsets is the same.

Time Complexity

The time complexity is `O(n * m)` where `n` is the number of elements in `nums` and `m` is half the sum of all elements in `nums` if the sum is even. This complexity arises from the double loop structure: an outer loop iterating over each number `x` in `nums`, and an inner loop iterating backwards from `m` to `x`. The inner loop runs at most `m` iterations (representing the possible sums up to half the total sum), and this is done for each of the `n` numbers.

Space Complexity

The space complexity is `O(m)` where `m` is half the sum of all elements in `nums` (if the sum is even). This is due to the array `f`, which stores Boolean values indicating whether a certain sum can be reached with the current subset of numbers. The array `f` has a length of `m + 1`, with `m` being the target sum (the zero is included to represent the empty subset).