Partition Equal Subset Sum Problem: Documentation

1. Problem Statement

Given an integer array nums, return true if it is possible to partition the array into two subsets such that the sum of the elements in both subsets is equal. Otherwise, return false.

Example 1:

- **Input**: nums = [1, 5, 11, 5]
- **Output**: true
- **Explanation**: The array can be partitioned into [1, 5, 5] and [11], both having an equal sum.

Example 2:

- **Input**: nums = [1, 2, 3, 5]
- **Output**: false
- **Explanation**: The array cannot be partitioned into equal sum subsets.

2. Intuition

This problem is a variation of the **Subset Sum Problem**. To determine if the array can be partitioned into two subsets with equal sums, we first check if the total sum of all the elements in the array is even. If it is, we aim to find a subset of the array whose sum is exactly half of the total sum. If such a subset exists, the other elements will form the second subset with the same sum.

Key Insight:

- If the total sum of the array is odd, it is **impossible** to partition it into two subsets with equal sum.
- If the total sum is even, we need to check if we can find a subset with a sum equal to total_sum / 2.

3. Key Observations

1. **Even Total Sum**: The problem is solvable only if the total sum of the array is even. If it's odd, there is no way to partition it into two equal subsets.
2. **Subset Sum**: The task reduces to finding whether there exists a subset of numbers in the array that adds up to half the total sum.
3. **Dynamic Programming Approach**: The problem can be efficiently solved using dynamic programming, specifically by using a **1D DP array** where each index represents whether a subset sum is achievable.
4. **Subset Sum Problem**: This is a classical problem, and by solving it, we can determine if partitioning into equal subsets is possible.

4. Approach

Dynamic Programming:

1. **Step 1**: Compute the total sum of the elements in nums. If the sum is odd, return false because it can't be split equally.
2. **Step 2**: Compute target = total_sum / 2. We now need to check if there exists a subset of nums whose sum is equal to target.
3. **Step 3**: Use a DP array dp of size (target + 1) where dp[i] represents whether a sum of i can be formed from any subset of nums. Initialize dp[0] as True since a sum of 0 is always achievable (by taking no elements).
4. **Step 4**: For each number in nums, update the DP array by iterating backward from target down to the current number. For each i, check if dp[i - num] is True, which means that i can be formed by adding num to a previously achievable sum.
5. **Step 5**: Finally, if dp[target] is True, it means that a subset with sum target exists, and hence the array can be partitioned into two subsets with equal sum.

5. Edge Cases

1. **Empty Array**: An empty array can trivially be partitioned into two empty subsets with equal sum. Return true.
2. **Single Element**: If the array has only one element, it is impossible to partition it into two non-empty subsets. Return false.
3. **Odd Total Sum**: If the total sum of the array is odd, return false immediately.
4. **Large Arrays**: Arrays with a large number of elements (up to the given constraint) should be handled efficiently by the dynamic programming solution to avoid time limits.

6. Complexity Analysis

Time Complexity:

- **Time Complexity**: O(n * target), where n is the number of elements in the array and target is the sum we are trying to achieve (i.e., total_sum / 2). For each number in nums, we update the DP array which takes target iterations.

Space Complexity:

- **Space Complexity**: O(target), because we are using a 1D DP array to store possible subset sums up to target. This space usage is optimal and avoids the need for a 2D DP array.

7. Alternative Approaches

1. **Backtracking**: One could use backtracking to explore all possible subsets, but this would be inefficient with time complexity O(2^n), which is not feasible for larger arrays.
2. **Greedy Approach**: Sorting the array and trying to assign numbers to subsets based on their values might seem like an option, but it doesn't guarantee the solution and is not guaranteed to work for all cases.
3. **Meet in the Middle**: This approach splits the array into two halves, and finds subsets of each half that sum to half of the total sum. It is more complicated than the dynamic programming approach but can be faster for some cases.

8. Test Cases

Test Case 1:

```
nums = [1, 5, 11, 5]
solution = Solution()
print(solution.canPartition(nums))  # Expected Output: True
```

Test Case 2:

```
nums = [1, 2, 3, 5]
solution = Solution()
print(solution.canPartition(nums))  # Expected Output: False
```

Test Case 3:

```
nums = [1, 2, 3, 4, 5]
solution = Solution()
print(solution.canPartition(nums))  # Expected Output: True
```

Test Case 4:

```
nums = [1, 2]
solution = Solution()
print(solution.canPartition(nums))  # Expected Output: False
```

9. Final Thoughts

The **Partition Equal Subset Sum** problem is a classic dynamic programming challenge. The key idea is to reduce the problem to a subset sum problem and then efficiently solve it using a DP array. By understanding the problem, the algorithm can be optimized to work even with larger inputs. The time complexity of O(n * target) is manageable for input sizes within the given constraints, making this a highly effective solution.