Equal Sum Partition DP

equal sum partition explain everything and code it in c++

Equal Sum Partition Problem

The **Equal Sum Partition Problem** is a classic problem in dynamic programming. Here's the formal definition:

Problem Statement

Given an array of positive integers, determine if it can be partitioned into two subsets such that the sum of the elements in both subsets is the same.

Approach

Key Insight

If we can partition the array into two subsets with equal sum, then the sum of all elements in the array must be even. If the total sum is odd, partitioning into two subsets with equal sum is not possible.

Let the total sum of the array be `S`. The problem reduces to finding a subset whose sum is `S/2`.

Steps to Solve

1. **Check the sum of the array:**
   - Compute the sum of the array, `S`.
   - If `S` is odd, return `false` since it's impossible to divide the array into two subsets with equal sum.
2. **Subset Sum Problem:**
   - Check if there's a subset of the array whose sum equals `S/2`.
   - This is a standard subset sum problem, which can be solved using recursion, memoization, or dynamic programming.

Recursive Solution

Recursive Relation

Let `f(n, target)` represent whether it's possible to form a sum of `target` using the first `n` elements of the array.

The relation:

- If `arr[n-1] > target`: Exclude the element. $f(n, target) = f(n-1, target)$
- Otherwise: Either include or exclude the element. $f(n, target) = f(n-1, target)$ OR $f(n-1, target - arr[n-1])$

Base Cases

- $f(0, 0) = true$ (Empty subset can form sum 0)
- $f(0, target) = false$ for $target > 0$ (No elements to form the sum)

C++ Implementation

Here's a recursive solution with memoization:

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

// Memoization table
int dp[201][10001];

// Recursive function to check subset sum
bool subsetSum(vector<int>& arr, int n, int target) {
    // Base cases
    if (target == 0) return true;
    if (n == 0) return false;

    // Check if already computed
    if (dp[n][target] != -1) return dp[n][target];

    // Exclude the current element
    bool exclude = subsetSum(arr, n - 1, target);

    // Include the current element (if it doesn't exceed the target)
    bool include = false;
    if (arr[n - 1] <= target) {
        include = subsetSum(arr, n - 1, target - arr[n - 1]);
    }

    // Store and return the result
    return dp[n][target] = exclude || include;
}

// Main function to check equal sum partition
bool canPartition(vector<int>& arr) {
    int sum = 0;
    for (int num : arr) sum += num;

    // If total sum is odd, partition is not possible
    if (sum % 2 != 0) return false;

    // Initialize memoization table
    memset(dp, -1, sizeof(dp));

    // Check for subset sum with target sum/2
    return subsetSum(arr, arr.size(), sum / 2);
}

int main() {
    vector<int> arr = {1, 5, 11, 5};
    if (canPartition(arr)) {
        cout << "The array can be partitioned into two subsets with equal sum.\n";
    } else {
```

```
            cout << "The array cannot be partitioned into two subsets with equal sum.\n";
        }
        return 0;
    }
```

Explanation of the Code

1. **`subsetSum` Function**:
    - Uses recursion with memoization to determine if a subset with a given sum exists.
    - Base cases handle trivial situations (e.g., no elements or zero sum).
2. **`canPartition` Function**:
    - Calculates the total sum of the array.
    - If the sum is odd, immediately returns **`false`**.
    - Otherwise, invokes **`subsetSum`** to check for a subset with sum **`S/2`**.
3. **Memoization Table**:
    - The **`dp`** table stores intermediate results to avoid redundant calculations.

Time Complexity

- Recursive calls: $O(n \times target)$, where **`n`** is the number of elements and **`target`** is **`sum/2`**.
- Space complexity: $O(n \times target)$ for the memoization table.

Output Example

For the input array **`{1, 5, 11, 5}`**, the output will be:

```
sql


  The array can be partitioned into two subsets with equal sum.
```