Equal Sum Partition C++

equal sum problem explain all the method and give the code in c++

The **Equal Sum Partition Problem** involves determining whether a given array can be partitioned into two subsets such that the sum of elements in both subsets is equal. Here's a comprehensive explanation of all methods:

Problem Definition

Given an array `arr` of size `n`, the goal is to check if it can be partitioned into two subsets with an equal sum.

Key Insight:

If the total sum of the array is odd, partitioning into two equal subsets is **impossible**. Otherwise, the problem reduces to finding a subset with a sum equal to `total_sum / 2`.

Methods to Solve the Problem

1. Recursive Approach

This is the simplest approach. The idea is to try every possible subset and see if there exists one with the required sum.

Steps:

1. Calculate the total sum of the array.
2. If the total sum is odd, return `false`.
3. Use recursion to check if a subset with a sum equal to `total_sum / 2` exists.

Code:

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

bool canPartitionRecursive(int index, int target, const vector<int>& arr) {
    if (target == 0) return true; // Found a subset with the required sum
    if (index == 0) return arr[0] == target; // Base case

    // Exclude the current element
```

```
    bool exclude = canPartitionRecursive(index - 1, target, arr);

    // Include the current element (only if it does not exceed the target)
    bool include = false;
    if (target >= arr[index])
        include = canPartitionRecursive(index - 1, target - arr[index], arr);

    return include || exclude;
}

bool canPartition(const vector<int>& arr) {
    int totalSum = 0;
    for (int num : arr) totalSum += num;

    if (totalSum % 2 != 0) return false; // Odd total sum cannot be partitioned

    return canPartitionRecursive(arr.size() - 1, totalSum / 2, arr);
}

int main() {
    vector<int> arr = {1, 5, 11, 5};
    if (canPartition(arr))
        cout << "Can partition into equal subsets" << endl;
    else
        cout << "Cannot partition into equal subsets" << endl;

    return 0;
}
```

---

2. Dynamic Programming (Top-Down Approach with Memoization)

This approach optimizes the recursive solution by storing results of subproblems to avoid redundant calculations.

Steps:

1. Use a 2D `dp` array where `dp[i][j]` stores whether it's possible to achieve sum `j` using the first `i` elements.
2. Initialize the base cases.
3. Fill the `dp` table iteratively.

Code:

```
cpp


#include <iostream>
#include <vector>
using namespace std;

bool canPartitionRecursive(int index, int target, const vector<int>& arr, vector<vector<int>
>& dp) {
    if (target == 0) return true;
    if (index == 0) return arr[0] == target;

    if (dp[index][target] != -1) return dp[index][target];

    bool exclude = canPartitionRecursive(index - 1, target, arr, dp);
    bool include = false;
    if (target >= arr[index])
        include = canPartitionRecursive(index - 1, target - arr[index], arr, dp);

    return dp[index][target] = (include || exclude);
}
```

```
bool canPartition(const vector<int>& arr) {
    int totalSum = 0;
    for (int num : arr) totalSum += num;

    if (totalSum % 2 != 0) return false;

    int target = totalSum / 2;
    vector<vector<int>> dp(arr.size(), vector<int>(target + 1, -1));

    return canPartitionRecursive(arr.size() - 1, target, arr, dp);
}

int main() {
    vector<int> arr = {1, 5, 11, 5};
    if (canPartition(arr))
        cout << "Can partition into equal subsets" << endl;
    else
        cout << "Cannot partition into equal subsets" << endl;

    return 0;
}
```

---

3. Dynamic Programming (Bottom-Up Approach)

This approach builds the solution iteratively using a table.

Steps:

1. Create a `dp` array where `dp[j]` represents whether a subset with sum `j` is possible.
2. Initialize `dp[0]` to `true` (a sum of 0 is always achievable).
3. For each number in the array, update the `dp` array from back to front.

Code:

cpp

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

bool canPartition(const vector<int>& arr) {
    int totalSum = 0;
    for (int num : arr) totalSum += num;

    if (totalSum % 2 != 0) return false;

    int target = totalSum / 2;
    vector<bool> dp(target + 1, false);
    dp[0] = true;

    for (int num : arr) {
        for (int j = target; j >= num; --j) {
            dp[j] = dp[j] || dp[j - num];
        }
    }

    return dp[target];
}

int main() {
    vector<int> arr = {1, 5, 11, 5};
    if (canPartition(arr))
        cout << "Can partition into equal subsets" << endl;
```

```
        else
            cout << "Cannot partition into equal subsets" << endl;

        return 0;
    }
```

Complexity Comparison

Method	Time Complexity	Space Complexity
Recursive	O(2^n)	O(n) (recursion stack)
Top-Down DP	O(n * target)	O(n * target)
Bottom-Up DP	O(n * target)	O(target)

Summary

- **Recursive Approach**: Simple but inefficient for large inputs.
- **Top-Down DP**: Optimized recursion with memoization.
- **Bottom-Up DP**: Most efficient in terms of both time and space.