Count of Subset Sum

count of sumset sum explain all the mathods and code in c++

The **Count of Subset Sum** problem involves finding the number of subsets in an array whose elements sum to a given target value, `sum`. Below, I'll explain different methods to solve this problem, along with their respective C++ implementations.

Problem Statement

Given an array `arr` of size `n` and a target sum `S`, find the number of subsets in `arr` whose sum equals `S`.

Method 1: Recursive Approach (Brute Force)

Explanation

This method explores all possible subsets using recursion. For each element in the array:

- Include the element in the current subset.
- Exclude the element from the current subset.

Algorithm

1. Base Case:
   - If `sum == 0`, return 1 (empty subset counts as a valid subset).
   - If `n == 0` and `sum != 0`, return 0 (no subsets possible).
2. Recursive Case:
   - Include the current element and reduce the sum.
   - Exclude the current element and keep the sum unchanged.
3. Return the sum of the above two cases.

Code (Recursive)

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

int countSubsetsRecursive(int arr[], int n, int sum) {
    if (sum == 0) return 1;  // Found a valid subset
    if (n == 0) return 0;    // No elements left

    if (arr[n - 1] <= sum) {
        // Include the element and exclude the element
```

```cpp
        return countSubsetsRecursive(arr, n - 1, sum - arr[n - 1]) +
               countSubsetsRecursive(arr, n - 1, sum);
    } else {
        // Exclude the element
        return countSubsetsRecursive(arr, n - 1, sum);
    }
}

int main() {
    int arr[] = {1, 2, 3, 3};
    int sum = 6;
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "Count of subsets: " << countSubsetsRecursive(arr, n, sum) << endl;
    return 0;
}
```

Method 2: Recursive with Memoization

Explanation

The recursive approach has overlapping subproblems, so we can use **Dynamic Programming (Memoization)** to store results of previously solved subproblems to optimize the solution.

Algorithm

- Use a 2D `dp` table where `dp[i][j]` represents the count of subsets from the first `i` elements that sum up to `j`.

Code (Recursive with Memoization)

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

int countSubsetsMemo(int arr[], int n, int sum, vector<vector<int>> &dp) {
    if (sum == 0) return 1;
    if (n == 0) return 0;

    if (dp[n][sum] != -1) return dp[n][sum];

    if (arr[n - 1] <= sum) {
        dp[n][sum] = countSubsetsMemo(arr, n - 1, sum - arr[n - 1], dp) +
                     countSubsetsMemo(arr, n - 1, sum, dp);
    } else {
        dp[n][sum] = countSubsetsMemo(arr, n - 1, sum, dp);
    }
    return dp[n][sum];
}

int main() {
    int arr[] = {1, 2, 3, 3};
    int sum = 6;
    int n = sizeof(arr) / sizeof(arr[0]);
    vector<vector<int>> dp(n + 1, vector<int>(sum + 1, -1));
    cout << "Count of subsets: " << countSubsetsMemo(arr, n, sum, dp) << endl;
    return 0;
}
```

Method 3: Tabulation (Bottom-Up DP)

Explanation

Use a 2D `dp` array to iteratively fill in the number of subsets that achieve a given sum.

Steps

1. Create a `dp` table of size `(n+1) x (sum+1)`.
2. Initialize:
    - `dp[i][0] = 1` (one way to make sum 0, by taking no elements).
    - `dp[0][j] = 0` for `j > 0` (no way to make positive sum with zero elements).
3. Iterate over the array and fill the table.

Code (Tabulation)

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

int countSubsetsTabulation(int arr[], int n, int sum) {
    vector<vector<int>> dp(n + 1, vector<int>(sum + 1, 0));

    for (int i = 0; i <= n; i++) dp[i][0] = 1;

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= sum; j++) {
            if (arr[i - 1] <= j) {
                dp[i][j] = dp[i - 1][j - arr[i - 1]] + dp[i - 1][j];
            } else {
                dp[i][j] = dp[i - 1][j];
            }
        }
    }
    return dp[n][sum];
}

int main() {
    int arr[] = {1, 2, 3, 3};
    int sum = 6;
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "Count of subsets: " << countSubsetsTabulation(arr, n, sum) << endl;
    return 0;
}
```

Method 4: Space Optimized DP

Explanation

Instead of using a 2D `dp` array, we use a 1D array to store results. This reduces space complexity from $O(n \times \text{sum})$ to $O(\text{sum})$.

Code (Space Optimized)

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

int countSubsetsSpaceOptimized(int arr[], int n, int sum) {
    vector<int> dp(sum + 1, 0);
    dp[0] = 1;

    for (int i = 0; i < n; i++) {
        for (int j = sum; j >= arr[i]; j--) {
            dp[j] += dp[j - arr[i]];
        }
    }
    return dp[sum];
}

int main() {
    int arr[] = {1, 2, 3, 3};
    int sum = 6;
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "Count of subsets: " << countSubsetsSpaceOptimized(arr, n, sum) << endl;
    return 0;
}
```

Time and Space Complexities

Method	Time Complexity	Space Complexity
Recursive (Brute Force)	$O(2^n)$	$O(n)$
Recursive with Memoization	$O(n \times S)$	$O(n \times S)$
Tabulation (Bottom-Up)	$O(n \times S)$	$O(n \times S)$
Space Optimized DP	$O(n \times S)$	$O(S)$

Each method is suitable for different use cases, depending on the size of the input and the need for optimization.