# Target Sum Methods C++

target sum explain all the methods and code in c++

The **Target Sum** problem is a classic dynamic programming problem. The goal is to determine if a subset of a given array can sum up to a target value. Let's explore all the methods to solve this problem with explanations and corresponding C++ code.

---

### Problem Statement

Given an array of integers `arr` and a target sum `target`, determine if there exists a subset whose sum equals `target`.

---

### Methods to Solve the Problem

#### 1. Recursive Approach

The idea is to explore all subsets of the array using recursion and check if any subset sums to the target.

**Steps:**

1. For each element, you have two choices:
   - Include it in the subset.
   - Exclude it from the subset.
2. Recursively check both possibilities.
3. Base cases:
   - If `target == 0`, return `true` (subset found).
   - If `n == 0` and `target != 0`, return `false`.

**Code:**

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

bool targetSumRecursive(vector<int>& arr, int n, int target) {
    if (target == 0) return true;  // Subset found
    if (n == 0) return false;      // No elements left

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

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

    return exclude || include;
```

```
}

int main() {
    vector<int> arr = {2, 3, 7, 8, 10};
    int target = 11;
    cout << (targetSumRecursive(arr, arr.size(), target) ? "Yes" : "No") << endl;
    return 0;
}
```

---

### 2. Memoized (Top-Down) Approach

To optimize the recursive solution, use a memoization table to store results of subproblems.

**Steps:**

1. Use a 2D table **`dp[n][target]`** where **`n`** is the number of elements and **`target`** is the sum.
2. Store the result of each subproblem in **`dp`** to avoid redundant calculations.

**Code:**

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

bool targetSumMemo(vector<int>& arr, int n, int target, vector<vector<int>>& dp) {
    if (target == 0) return true;
    if (n == 0) return false;

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

    bool exclude = targetSumMemo(arr, n - 1, target, dp);
    bool include = (arr[n - 1] <= target) &&
                   targetSumMemo(arr, n - 1, target - arr[n - 1], dp);

    return dp[n][target] = exclude || include;
}

int main() {
    vector<int> arr = {2, 3, 7, 8, 10};
    int target = 11;
    int n = arr.size();
    vector<vector<int>> dp(n + 1, vector<int>(target + 1, -1));
    cout << (targetSumMemo(arr, n, target, dp) ? "Yes" : "No") << endl;
    return 0;
}
```

---

### 3. Dynamic Programming (Bottom-Up) Approach

The bottom-up approach builds the solution iteratively using a 2D DP table.

**Steps:**

1. Create a DP table **`dp[n+1][target+1]`**.
2. **`dp[i][j]`** is **`true`** if a subset of the first **`i`** elements has a sum **`j`**.
3. Base case:
    - **`dp[i][0] = true`** for all **`i`** (sum 0 is always achievable).
4. Fill the table iteratively.

**Code:**

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

bool targetSumDP(vector<int>& arr, int target) {
    int n = arr.size();
    vector<vector<bool>> dp(n + 1, vector<bool>(target + 1, false));

    // Base case: sum 0 is always achievable
    for (int i = 0; i <= n; i++) dp[i][0] = true;

    // Fill the DP table
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= target; j++) {
            dp[i][j] = dp[i - 1][j];  // Exclude the element
            if (arr[i - 1] <= j) {
                dp[i][j] = dp[i][j] || dp[i - 1][j - arr[i - 1]];  // Include the element
            }
        }
    }

    return dp[n][target];
}

int main() {
    vector<int> arr = {2, 3, 7, 8, 10};
    int target = 11;
    cout << (targetSumDP(arr, target) ? "Yes" : "No") << endl;
    return 0;
}
```

---

### 4. Space-Optimized DP

Since the DP table only depends on the previous row, we can optimize space to `O(target)`.

**Code:**

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

bool targetSumSpaceOptimized(vector<int>& arr, int target) {
    int n = arr.size();
    vector<bool> prev(target + 1, false), curr(target + 1, false);

    prev[0] = true;  // Base case

    for (int i = 1; i <= n; i++) {
        curr[0] = true;  // Sum 0 is always possible
        for (int j = 1; j <= target; j++) {
            curr[j] = prev[j];  // Exclude the element
            if (arr[i - 1] <= j) {
                curr[j] = curr[j] || prev[j - arr[i - 1]];  // Include the element
            }
        }
        prev = curr;  // Update previous row
    }

    return prev[target];
```

```
}

int main() {
    vector<int> arr = {2, 3, 7, 8, 10};
    int target = 11;
    cout << (targetSumSpaceOptimized(arr, target) ? "Yes" : "No") << endl;
    return 0;
}
```

---

### Comparison of Methods

| Method | Time Complexity | Space Complexity |
|---|---|---|
| Recursive | $O(2^n)$ | $O(n)$ |
| Memoized | $O(n \times target)$ | $O(n \times target)$ |
| Dynamic Programming (DP) | $O(n \times target)$ | $O(n \times target)$ |
| Space-Optimized DP | $O(n \times target)$ | $O(target)$ |

---

Choose the method based on the size of the input and the need for optimization. Let me know if you need further clarification or more examples!