Subset Sum Problem CPP

subset sum problem explain every thing and give code in c++

Subset Sum Problem

The **Subset Sum Problem** is a classic problem in computer science and dynamic programming. It is defined as follows:

Problem Statement:

Given a set of non-negative integers and a value S, determine if there is a subset of the given set whose sum equals S.

Example:

1. **Input:**
   Set = {3, 34, 4, 12, 5, 2}, $S = 9$
   Output: Yes (Subset: {4, 5})
2. **Input:**
   Set = {3, 34, 4, 12, 5, 2}, $S = 30$
   Output: No

Approach to Solve the Problem

There are two primary methods to solve the Subset Sum Problem:

1. Recursive Approach

This method involves trying every possible subset to check if the subset's sum equals S.

Recursive Formula:

- Let $A[i]$ be the i-th element in the set, and $isSubsetSum(n, S)$ be a function that returns `` `true` `` if there exists a subset of the first n elements that has a sum S.

  $isSubsetSum(n, S) = isSubsetSum(n-1, S)$ or $isSubsetSum(n-1, S - A[n-1])$

Explanation:

- **Include the element:** Check $isSubsetSum(n-1, S - A[n-1])$.
- **Exclude the element:** Check $isSubsetSum(n-1, S)$.

2. Dynamic Programming Approach

Use a 2D boolean array $dp[n+1][S+1]$ where:

- $dp[i][j]$ is `true` if a subset of the first i elements has a sum of j.
- The solution is found in $dp[n][S]$.

Transition Relation:

- $dp[i][j] = dp[i-1][j]$ (excluding $A[i-1]$)
- $dp[i][j] = dp[i-1][j] \,||\, dp[i-1][j-A[i-1]]$ (including $A[i-1]$)

Base Cases:

- $dp[i][0] = true$ for all i (a sum of 0 can be achieved with an empty subset).
- $dp[0][j] = false$ for all $j > 0$ (no subset can achieve a non-zero sum if there are no elements).

C++ Implementation

1. Recursive Solution

cpp

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

bool isSubsetSum(int arr[], int n, int sum) {
    // Base cases
    if (sum == 0) return true;
    if (n == 0) return false;

    // If last element is greater than sum, ignore it
    if (arr[n-1] > sum)
        return isSubsetSum(arr, n-1, sum);

    // Check if sum can be obtained by including or excluding the last element
    return isSubsetSum(arr, n-1, sum) || isSubsetSum(arr, n-1, sum - arr[n-1]);
}

int main() {
    int arr[] = {3, 34, 4, 12, 5, 2};
    int sum = 9;
    int n = sizeof(arr) / sizeof(arr[0]);

    if (isSubsetSum(arr, n, sum))
        cout << "Yes, a subset with the given sum exists.\n";
    else
        cout << "No, such a subset does not exist.\n";

    return 0;
}
```

2. Dynamic Programming Solution

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

bool isSubsetSum(int arr[], int n, int sum) {
    vector<vector<bool>> dp(n+1, vector<bool>(sum+1, false));

    // Base cases
    for (int i = 0; i <= n; ++i)
        dp[i][0] = true; // Sum 0 is possible with an empty subset

    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] || dp[i-1][j - arr[i-1]];
            else
                dp[i][j] = dp[i-1][j];
        }
    }

    return dp[n][sum];
}

int main() {
    int arr[] = {3, 34, 4, 12, 5, 2};
    int sum = 9;
    int n = sizeof(arr) / sizeof(arr[0]);

    if (isSubsetSum(arr, n, sum))
        cout << "Yes, a subset with the given sum exists.\n";
    else
        cout << "No, such a subset does not exist.\n";

    return 0;
}
```

Time Complexity:

1. **Recursive Solution:** $O(2^n)$ (Exponential due to all subset combinations).
2. **Dynamic Programming Solution:** $O(n \times S)$, where n is the size of the set and S is the sum.

Space Complexity:

1. **Recursive Solution:** $O(n)$ (function call stack).
2. **Dynamic Programming Solution:** $O(n \times S)$ (for the DP table).

all the methode in db recursive and memoization and tabulation

Subset Sum Problem - All Methods in Detail

Here's a breakdown of the **Subset Sum Problem** using the following approaches:

1. **Recursive Approach (Without Memoization)**

2. **Recursive Approach with Memoization**
3. **Dynamic Programming (Tabulation)**

---

1. Recursive Approach (Without Memoization)

Explanation:

This is a brute-force method where we explore all possible subsets to check if any of them sum to the target S.
We decide for each element:

- Include it in the subset.
- Exclude it from the subset.

Recursive Relation:

Let $isSubsetSum(n, S)$ denote whether a subset with sum S can be formed using the first n elements:

- $isSubsetSum(n, S) = isSubsetSum(n-1, S)$ OR $isSubsetSum(n-1, S - arr[n-1])$

Base Cases:

- If $S = 0$: Return `true` (an empty subset always sums to 0).
- If $n = 0$ and $S > 0$: Return `false` (no subset can be formed).

C++ Code:

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

bool isSubsetSum(int arr[], int n, int sum) {
    // Base cases
    if (sum == 0) return true;
    if (n == 0) return false;

    // Ignore the last element if it's greater than the remaining sum
    if (arr[n-1] > sum)
        return isSubsetSum(arr, n-1, sum);

    // Check including or excluding the last element
    return isSubsetSum(arr, n-1, sum) || isSubsetSum(arr, n-1, sum - arr[n-1]);
}

int main() {
    int arr[] = {3, 34, 4, 12, 5, 2};
    int sum = 9;
    int n = sizeof(arr) / sizeof(arr[0]);

    if (isSubsetSum(arr, n, sum))
        cout << "Yes, a subset with the given sum exists.\n";
    else
        cout << "No, such a subset does not exist.\n";
```

```
    return 0;
}
```

Time Complexity: $O(2^n)$
Space Complexity: $O(n)$ (due to recursive stack)

---

2. Recursive Approach with Memoization

Explanation:

To optimize the recursive solution, we use **memoization** to store the results of previously computed subproblems. This avoids redundant calculations.

Memoization Table:

We use a 2D table `dp[n][sum]` where:

- `dp[i][j]` stores whether a subset of the first i elements can achieve the sum j.

Recursive Relation:

Same as before:

- $dp[n][S] = dp[n-1][S]$ OR $dp[n-1][S - arr[n-1]]$

C++ Code:

cpp

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

bool isSubsetSum(int arr[], int n, int sum, vector<vector<int>> &dp) {
    // Base cases
    if (sum == 0) return true;
    if (n == 0) return false;

    // If the result is already computed
    if (dp[n][sum] != -1)
        return dp[n][sum];

    // Ignore the last element if it's greater than the remaining sum
    if (arr[n-1] > sum)
        return dp[n][sum] = isSubsetSum(arr, n-1, sum, dp);

    // Check including or excluding the last element
    return dp[n][sum] = isSubsetSum(arr, n-1, sum, dp) || isSubsetSum(arr, n-1, sum - arr[n-1], dp);
}

int main() {
    int arr[] = {3, 34, 4, 12, 5, 2};
    int sum = 9;
    int n = sizeof(arr) / sizeof(arr[0]);

    // Initialize memoization table with -1
    vector<vector<int>> dp(n+1, vector<int>(sum+1, -1));
```

```
    if (isSubsetSum(arr, n, sum, dp))
        cout << "Yes, a subset with the given sum exists.\n";
    else
        cout << "No, such a subset does not exist.\n";

    return 0;
}
```

Time Complexity: $O(n \times S)$
Space Complexity: $O(n \times S)$ (for the memoization table)

---

3. Dynamic Programming (Tabulation)

Explanation:

This is a **bottom-up approach** where we iteratively fill a DP table to solve the problem.

DP Table:

Let `dp[i][j]` represent whether a subset of the first i elements can achieve a sum j.

Transition Relation:

- $dp[i][j] = dp[i-1][j]$ (excluding $arr[i-1]$)
- $dp[i][j] = dp[i-1][j] \,||\, dp[i-1][j - arr[i-1]]$ (including $arr[i-1]$)

Base Cases:

- $dp[i][0] = true$ for all i (a sum of 0 can always be achieved).
- $dp[0][j] = false$ for $j > 0$ (no subset can be formed with 0 elements).

C++ Code:

```
cpp


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

bool isSubsetSum(int arr[], int n, int sum) {
    // Create a DP table
    vector<vector<bool>> dp(n+1, vector<bool>(sum+1, false));

    // Base cases
    for (int i = 0; i <= n; ++i)
        dp[i][0] = true; // Sum 0 can always be achieved

    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] || dp[i-1][j - arr[i-1]];
            else
                dp[i][j] = dp[i-1][j];
        }
    }

    return dp[n][sum];
}
```

```
int main() {
    int arr[] = {3, 34, 4, 12, 5, 2};
    int sum = 9;
    int n = sizeof(arr) / sizeof(arr[0]);

    if (isSubsetSum(arr, n, sum))
        cout << "Yes, a subset with the given sum exists.\n";
    else
        cout << "No, such a subset does not exist.\n";

    return 0;
}
```

Time Complexity: $O(n \times S)$
Space Complexity: $O(n \times S)$

Comparison of Methods

Method	Time Complexity	Space Complexity	Remarks
Recursive	$O(2^n)$	$O(n)$	Simple but inefficient
Recursive + Memoization	$O(n \times S)$	$O(n \times S)$	Optimized with stored results
Dynamic Programming	$O(n \times S)$	$O(n \times S)$	Most efficient, iterative approach