Problem Statement

Given a set of positive numbers, find the total number of subsets whose sum is equal to a given number 'S'.

Example 1:

```
Input: {1, 1, 2, 3} S=4
Output: 3
The given set has '3' subsets whose sum is '4': {1, 1, 2}, {1, 3}, {1, 3}
Note that we have two similar sets {1, 3}, because we have two '1' in our
input.
```

Example 2:

```
Input: {1, 2, 7, 1, 5} S=9
Output: 3
The given set has '3' subsets whose sum is '9': {2, 7}, {1, 7, 1}, {1, 2,
1, 5}
```

Basic Solution

The problem follows the **0/1 Knapsack pattern** and is quite similar to Subet Sum. The only difference in this problem is that we need to count the number of subsets, whereas in the Subset Sum we only wanted to know if there exists a subset with the given sum.

A basic brute-force solution could be to try all subsets of the given numbers to count the subsets that have a sum equal to 'S'. So, our brute-force algorithm will look like:

```
for each number 'i':
    create a new set which includes number 'i' if it does not exceed 'S',
and recursively process the remaining numbers and sum
    create a new set without number 'i', and recursively process the
remaining numbers
return the count of subsets who has a sum equal to 'S'
```

Code:

```
const countSubsets = (num, sum) => {
    let countSubsetsHelper = (num, sum, currentIndex) => {
        // base checks
        if (sum === 0) return 1;

        if (num.length === 0 || currentIndex >= num.length) return 0;

        // recursive call after selecting the number at the currentIndex
```

```
        // if the number at currentIndex exceeds the sum, we shouldn't
process this
        let sum1 = 0;
        if (num[currentIndex] <= sum) {
            sum1 = countSubsetsHelper(num, sum - num[currentIndex],
currentIndex + 1);
        }

        // recursive call after excluding the number at the currentIndex
        const sum2 = countSubsetsHelper(num, sum, currentIndex + 1);
        return sum1 + sum2;
    };

    return countSubsetsHelper(num, sum, 0);
};

console.log(`Count of subset sum is: ---> ${countSubsets([1, 1, 2, 3],
4)}`);
console.log(`Count of subset sum is: ---> ${countSubsets([1, 2, 7, 1, 5],
9)}`);
```

The time complexity of the above algorithm is exponential O(2^n), where 'n' represents the total number. The space complexity is O(n), this memory is used to store the recursion stack.

Top-down Dynamic Programming with Memoization

We can use memoization to overcome the overlapping sub-problems. We will be using a two-dimensional array to store the results of solved sub-problems. As mentioned above, we need to store results for every subset and for every possible sum.

Code:

```
const countSubsets = (num, sum) => {
    const dp = [];
    let countSubsetsHelper = (num, sum, currentIndex) => {
        // base checks
        if (sum === 0) return 1;

        if (num.length === 0 || currentIndex >= num.length) return 0;

        dp[currentIndex] = dp[currentIndex] || [];

        if (typeof dp[currentIndex][sum] === 'undefined') {
            // recursive call after selecting the number at the
currentIndex
            // if the number at currentIndex exceeds the sum, we shouldn't
process this
            let sum1 = 0;
            if (num[currentIndex] <= sum) {
                sum1 = countSubsetsHelper(num, sum - num[currentIndex],
currentIndex + 1);
```

```
            }

            // recursive call after excluding the number at the
currentIndex
            const sum2 = countSubsetsHelper(num, sum, currentIndex + 1);

            dp[currentIndex][sum] = sum1 + sum2;
        }

        return dp[currentIndex][sum];
    };

    return countSubsetsHelper(num, sum, 0);
};

console.log(`Count of subset sum is: ---> ${countSubsets([1, 1, 2, 3],
4)}`);
console.log(`Count of subset sum is: ---> ${countSubsets([1, 2, 7, 1, 5],
9)}`);
```

Bottom-up Dynamic Programming

We will try to find if we can make all possible sums with every subset to populate the array `dp[totalNumbers][S+1]`

So, at every step we have two options:

1. Exclude the number. Count all the subsets without the given number up to the given sum => `dp[index-1][sum]`
2. Include the number if its value is not more than the 'sum'. In this case, we will count all the subsets to get the remaining sum => `dp[index-1][sum-num[index]]`

To find the total sets, we will add both of the above values: `dp[index][sum] = dp[index-1][sum] + dp[index-1][sum-num[index]]`

Code:

```
let countSubsets = function (num, sum) {
    const n = num.length;
    const dp = Array(n)
        .fill(0)
        .map(() => Array(sum + 1).fill(0));

    // populate the sum=0 columns, as we will always have an empty set for
zero sum
    for (let i = 0; i < n; i++) {
        dp[i][0] = 1;
    }

    // with only one number, we can form a subset only when the required
sum is equal to its value
```

```
    for (let s = 1; s <= sum; s++) {
        dp[0][s] = num[0] == s ? 1 : 0;
    }

    // process all subsets for all sums
    for (let i = 1; i < num.length; i++) {
        for (let s = 1; s <= sum; s++) {
            // exclude the number
            dp[i][s] = dp[i - 1][s];
            // include the number, if it does not exceed the sum
            if (s >= num[i]) {
                dp[i][s] += dp[i - 1][s - num[i]];
            }
        }
    }

    // the bottom-right corner will have our answer.
    return dp[num.length - 1][sum];
};

console.log(`Count of subset sum is: ---> ${countSubsets([1, 1, 2, 3],
4)}`);
console.log(`Count of subset sum is: ---> ${countSubsets([1, 2, 7, 1, 5],
9)}`);
```

The above solution has time and space complexity of `O(N*S)`, where 'N' represents total numbers and 'S' is the desired sum.

Optimized

This has a space complexity of `O(S)`

```
const countSubsets = function (num, sum) {
    const n = num.length;
    const dp = Array(sum + 1).fill(0);
    dp[0] = 1;

    // with only one number, we can form a subset only when the required
sum is equal to its value
    for (let s = 1; s <= sum; s++) {
        dp[s] = num[0] == s ? 1 : 0;
    }

    // process all subsets for all sums
    for (let i = 1; i < num.length; i++) {
        for (let s = sum; s >= 0; s--) {
            if (s >= num[i]) {
                dp[s] += dp[s - num[i]];
            }
        }
    }
```

```
    return dp[sum];
};
console.log(`Count of subset sum is: ---> ${countSubsets([1, 1, 2, 3],
4)}`);
console.log(`Count of subset sum is: ---> ${countSubsets([1, 2, 7, 1, 5],
9)}`);
```