Problem Statement

Given a set of positive numbers, determine if there exists a subset whose sum is equal to a given number 'S'

Example1:

```
Input: {1, 2, 3, 7} S=6
Output: True
The given set has a subset whose sum is '6': {1, 2, 3}
```

Example2:

```
Input: {1, 2, 7, 1, 5} S=10
Output: True
The given set has a subset whose sum is '10': {1, 2, 7}
```

Example: 3

```
Input: {1, 3, 4, 8} S=6
Output: False
The given set does not have any subset whose sum is equal to '6'
```

Basic Solution

The problem follows the **0/1 Knapsack pattern** and its quite similar to Equal subset Sum partition. A basic brute-force solution could be to try all subsets of the given numbers to see if any set has a sum equal to 'S'

So our brute-force algorithm would be 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
    create a new set WITHOUT number 'i', and recursively process the
remaining numbers

return true if any of the above two sets has a sum equal to 'S', otherwise
return false
```

Since this problem is quite similar to Equal Subset Sum Partition, let's jump directly to the bottom-up dynamic programming solution

Bottom-up Dynamic Programming

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

For every possible sum 's' (where 0 <= s <= S), we have two options:

1. Exclude the number. In this case, we will see if we can get the sum 's' from the subset excluding this number => `dp[index-1][s]`
2. Include the number if it's value is not more than 's'. In this case, we will see if we can find a subset to get the remaining sum. `dp[index-1][s-num[index]]`

If either of the above two scenarios returns true, we can find a subset with a sum equal to 's'.

```
let subsetSumBottomUp = function (num, sum) {
    const n = num.length;

    const dp = Array(n)
        .fill(false)
        .map(() => Array(sum + 1).fill(false));

    // populate the sum=0 column, as we can always have 0 sum without any
element.
    for (let i = 0; i < n; i++) dp[i][0] = true;

    // 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;
    }

    // process all subsets for all sums
    for (let i = 1; i < n; i++) {
        for (let s = 1; s <= sum; s++) {
            // if we can get the sum 's' without the number at index 'i'
            if (dp[i - 1][s]) {
                dp[i][s] = dp[i - 1][s];
            } else if (s >= num[i]) {
                // else include the number and see if we can find a subset
to get the remaining sum
                dp[i][s] = dp[i - 1][s - num[i]];
            }
        }
    }

    return dp[n - 1][sum];
};

console.log(`Can partitioning be done: ---> ${subsetSumBottomUp([1, 2, 3,
4], 6)}`);
console.log(`Can partitioning be done: ---> ${subsetSumBottomUp([1, 2, 7,
1, 5], 10)}`);
console.log(`Can partitioning be done: ---> ${subsetSumBottomUp([1, 3, 4,
8], 6)}`);
```

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

Optimization

Can we further improve our bottom-up DP solution? Can you find an algorithm that has O(S) space complexity?

```
let subsetSumBottomUpOptimized = function (num, sum) {
    const n = num.length;

    const dp = Array(sum + 1).fill(false);

    // populate the sum=0 column, as we can always have 0 sum without any
element.
    dp[0] = true;

    // 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;
    }

    // process all subsets for all sums
    for (let i = 1; i < n; i++) {
        for (let s = sum; s >= 0; s--) {
            // if dp[s]==true, this means we can get the sum 's' without
num[i], hence we can move on to
            // the next number else we can include num[i] and see if we
can find a subset to get the
            // remaining sum
            if (!dp[s] && s >= num[i]) {
                dp[s] = dp[s - num[i]];
            }
        }
    }

    return dp[sum];
};

console.log(`Can partitioning be done: --->
${subsetSumBottomUpOptimized([1, 2, 3, 4], 6)}`);
console.log(`Can partitioning be done: --->
${subsetSumBottomUpOptimized([1, 2, 7, 1, 5], 10)}`);
console.log(`Can partitioning be done: --->
${subsetSumBottomUpOptimized([1, 3, 4, 8], 6)}`);
```