# Subset Sum (medium)



## Problem Statement

Given a set of positive numbers, determine if a subset exists whose sum is equal to a given number ‘S’.

Example 1:

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

Example 2:

```
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

This problem follows the **0/1 Knapsack pattern** and is 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 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
  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

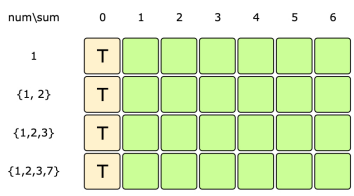
We’ll try to find if we can make all possible sums with every subset to populate the array `dp[TotalNumbers][S+1]`.

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 its 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’.

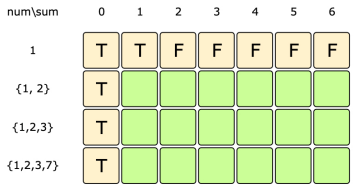
Let’s draw this visually, with the example input {1, 2, 3, 7}, and start with our base case of size zero:



Mark Course as Completed

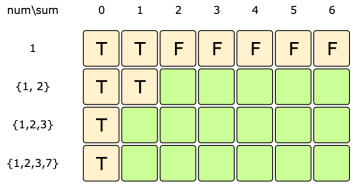
'0' sum can always be found through an empty set

**1** of 10



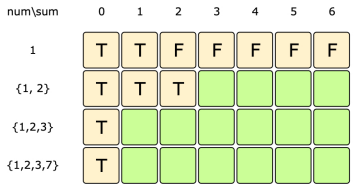

With only one number, we can form a subset only when the required sum is equal to that number

**2** of 10



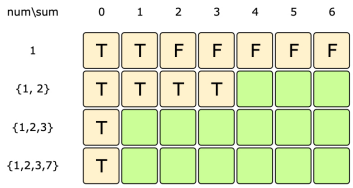

sum: 1, index:1=> (dp[index-1][sum] , as the 'sum' is less than the number at index '1' (i.e., 1 < 2)

**3** of 10



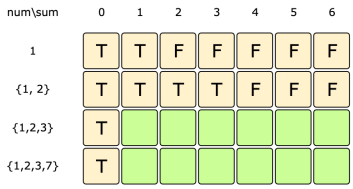

sum: 2, index:1=> (dp[index-1][sum] || dp[index-1][sum-2])

**4** of 10



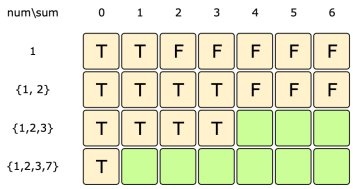

sum: 3, index:1=> (dp[index-1][sum] || dp[index-1][sum-2])

**5** of 10



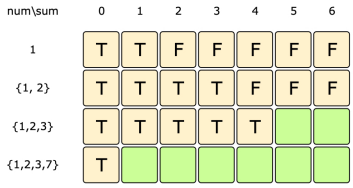

sum: 4-6 index:1=> (dp[index-1][sum] || dp[index-1][sum-2])

**6** of 10



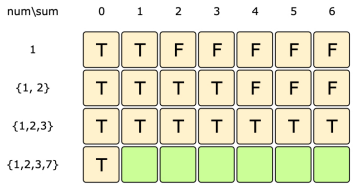

sum: 1,2,3, index:2=> (dp[index-1][sum] || dp[index-1][sum-3])

**7** of 10



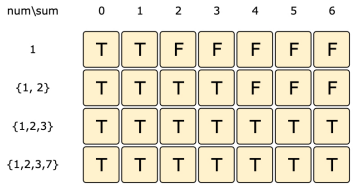

sum: 4, index:2=> (dp[index-1][sum] || dp[index-1][sum-3])

**8** of 10




sum: 5, 6, index:2=> (dp[index-1][sum] || dp[index-1][sum-3])

**9** of 10




sum: 1-6, index:3=> (dp[index-1][sum] || dp[index-1][sum-7])

**10** of 10

## Code

Here is the code for our bottom-up dynamic programming approach:

Java | Python3 | C++ | JS

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

  // populate the sum=0 columns, as we can always form '0' sum with an empty set
  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 < num.length; 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]];
      }
    }
  }

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

RUN SAVE RESET

Close

Output 1.557s

```
true
true
false
```

## Time and Space complexity

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

## Challenge

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

Show Hint

Java | Python3 | C++ | JS

```
const canPartition = function (num, sum) {
  const n = num.length;
  const dp = Array(sum + 1).fill(false);

  // handle sum=0, as we can always have '0' sum with an empty set
  dp[0] = true;

  // with only one number, we can have 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: ---> ${canPartition([1, 2, 3, 4], 6)}`);
```

RUN | SAVE | RESET

Close

Output 1.675s

```
Can partitioning be done: ---> true
Can partitioning be done: ---> true
Can partitioning be done: ---> false
```



Back
Equal Subset Sum Partition (medium)

Next
Minimum Subset Sum Difference (hard)

Mark as Completed

Report an Issue | Ask a Question

Explore
Tracks
My Courses
Edpresso
Refer a Friend
Create