Equal Subset Sum Partition

Problem Statement

Given a set of positive numbers, find if we can partition it into two subsets such that the sum of elements in both the subsets are equal.

Example 1:

```
Input: {1, 2, 3, 4}
Output: True
Explanation: The given set can be partitioned into two subsets with equal
sum: {1, 4} and {2, 3}
```

Example 2:

```
Input: {1, 1, 3, 4, 7}
Output: True
Explanation: The given set can be partitioned into two subsets with equal
sum: {1, 3, 4} and {1, 7}
```

Example 3:

```
Input: {2, 3, 4, 6}
Output: False
Explanation: The given set cannot be partitioned into two subsets with
equal sum.
```

The problem looks similar to the 0/1 Knapsack problem. Try to solve it first!

Basic Solution

The problem follows the **0/1 Knapsack pattern**. A basic brute-force solution could be to try all combinations of partitioning the given numbers into two sets to see if any pair of sets have equal sum.

Assume is `S` represents the total sum of all the given numbers, then the two equal subsets must have a sum equal to `S/2`. This essentially transforms our problem to: "Find a subset of the given numbers that has a total sum of `S/2`".

So our brute-force algorithm will look like:

```
for each number 'i':
    create a new set which INCLUDES 'i' if it does not exceed 'S/2', and
recursively process the remaining numbers.
```

```
    create a new set WITHOUT number 'i', and recursively process the
remaining items

return true if any of the above set has a sum equal to 'S/2', otherwise
return false;
```

Code:

```
let canPartitionRecursive = function (num) {
    let sum = 0;
    for (let i = 0; i < num.length; i++) sum += num[i];

    // if sum is an odd number we can't have two subsets with equal sum
    if (sum % 2 !== 0) return false;

    let canPartitionRecursiveHelper = function (sum, currentIndex) {
        // base check
        if (sum === 0) return true;

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

        // recursive call after choosing the number at the currentIndex
        // if the number at currentIndex exceeds the sum, we shouldn't
process this
        if (num[currentIndex] <= sum) {
            if (canPartitionRecursiveHelper(sum - num[currentIndex],
currentIndex + 1)) return true;
        }

        // recursive call after excluding the number at the currentIndex
        return canPartitionRecursiveHelper(sum, currentIndex + 1);
    };

    return canPartitionRecursiveHelper(sum / 2, 0);
};

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

he 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 which will be used to store the recursion stack.

Top-Down Dynamic Programming with Memoization

We can use memoization to overcome the overlapping sub-problems.

Since we need to store the results for every subset and for every possible sum, therefore we will be using a two-dimensional array to store the results of the solved sub-problems. The first dimension of the array will represent different subsets and the second dimension will represent different 'sums' that we can calculate from each subset. These two dimensions of the array can also be inferred from the two changing values (sum and currentIndex) in our recursive function `canPartitionRecursiveHelper()`

```
let canPartitionTopDown = function (num) {
    let sum = 0;
    for (let i = 0; i < num.length; i++) sum += num[i];

    // if sum is an odd number we can't have two subsets with equal sum
    if (sum % 2 !== 0) return false;

    const dp = [];

    let canPartitionTopDownHelper = function (sum, currentIndex) {
        // base check
        if (sum === 0) return true;

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

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

        if (dp[currentIndex][sum]) return dp[currentIndex][sum];

        // recursive call after choosing the number at the currentIndex
        // if the number at currentIndex exceeds the sum, we shouldn't
process this
        if (num[currentIndex] <= sum) {
            if (canPartitionTopDownHelper(sum - num[currentIndex],
currentIndex + 1)) {
                dp[currentIndex][sum] = true;
                return true;
            }
        }

        // recursive call after excluding the number at the currentIndex
        dp[currentIndex][sum] = canPartitionTopDownHelper(sum,
currentIndex + 1);
        return dp[currentIndex][sum];
    };

    return canPartitionTopDownHelper(sum / 2, 0);
};

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

```
console.log(`Can partitioning be done: ---> ${canPartitionTopDown([8, 8,
2])}`);
```

The above algorithm has time and space complexity of `O(N*S)`, where ‘N’ represents total numbers and ‘S’ is the total sum of all the numbers.

Bottom-up Dynamic Programming

Let's try to populate our `dp[][]` array from the above solution, working in a bottom-up fashion. Essentially, we want to find out if we can make all possible sums with every subset. **This means, dp[i][s] will be 'true' if we can make sum 's' from the firstt 'i' numbers.**

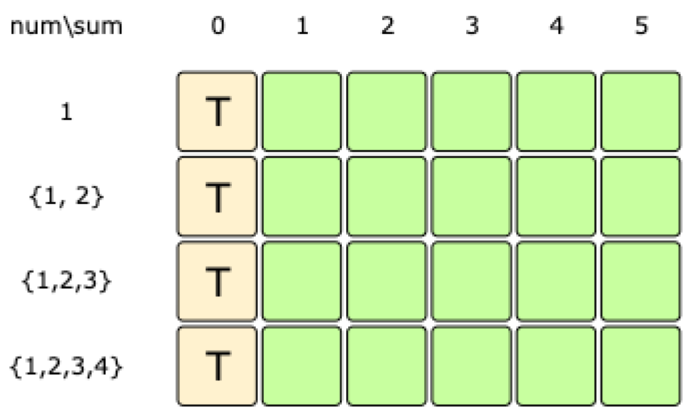
So, for each number at index 'i' and sum 's', we have two options:

1. Exclude the number. In this case, we will see if we can get ‘s’ from the subset excluding this number: `dp[i-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[i-1][s-num[i]]`

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

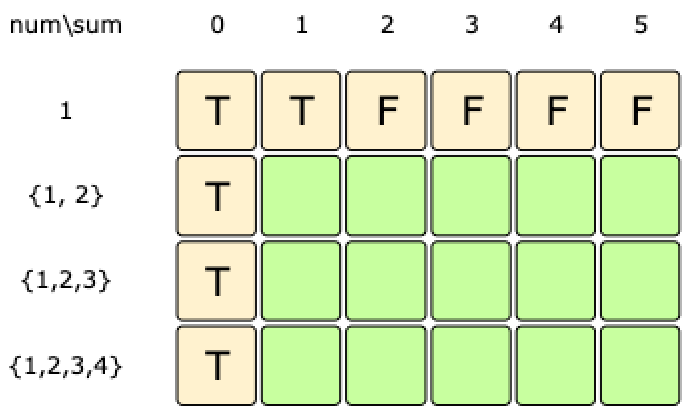
Let's start with our base case of zero capacity:

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



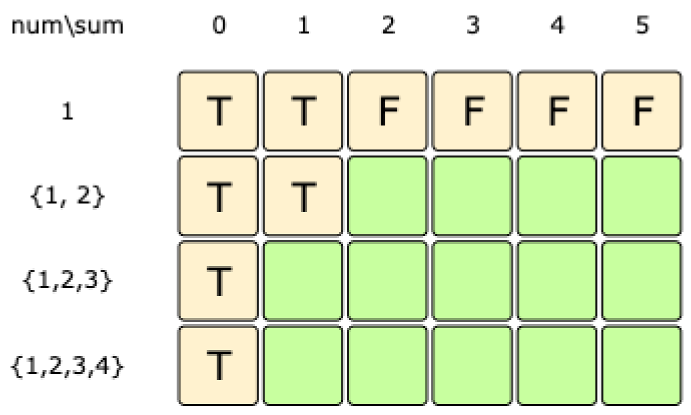
num\sum	0	1	2	3	4	5
1	T					
{1, 2}	T					
{1,2,3}	T					
{1,2,3,4}	T					

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



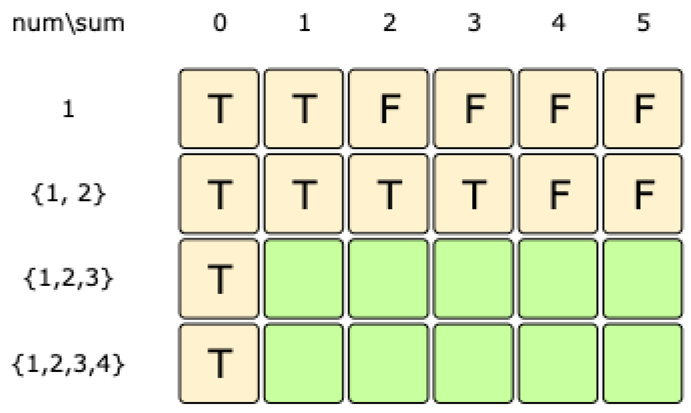
num\sum	0	1	2	3	4	5
1	T	T	F	F	F	F
{1, 2}	T					
{1,2,3}	T					
{1,2,3,4}	T					

`sum: 1, index: 1 => (dp[index-1][sum])`, as the 'sum' is less than the number at index '1'



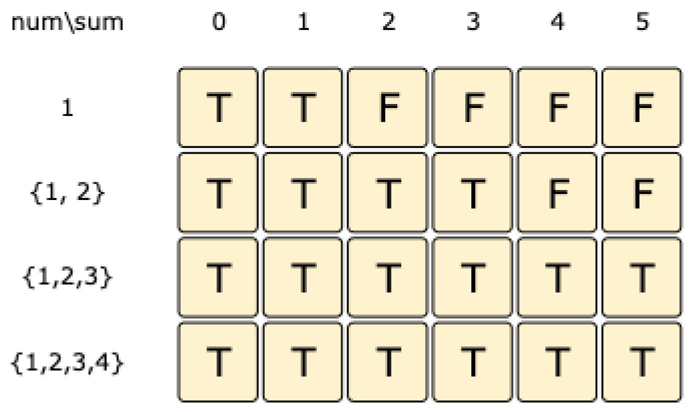
num\sum	0	1	2	3	4	5
1	T	T	F	F	F	F
{1, 2}	T	T	🟩	🟩	🟩	🟩
{1,2,3}	T	🟩	🟩	🟩	🟩	🟩
{1,2,3,4}	T	🟩	🟩	🟩	🟩	🟩

`sum: 4,5, index: 1 => (dp[index-1][sum] || dp[index-1][sum-2])`,



num\sum	0	1	2	3	4	5
1	T	T	F	F	F	F
{1, 2}	T	T	T	T	F	F
{1,2,3}	T	🟩	🟩	🟩	🟩	🟩
{1,2,3,4}	T	🟩	🟩	🟩	🟩	🟩

`sum: 1-5, index:3=> (dp[index-1][sum] || dp[index-1][sum-4])`



num\sum	0	1	2	3	4	5
1	T	T	F	F	F	F
{1, 2}	T	T	T	T	F	F
{1,2,3}	T	T	T	T	T	T
{1,2,3,4}	T	T	T	T	T	T

From the above visualizations, we can clearly see that it is possible to partition the given set into two subsets with equal sums, as shown by the bottom right cell `dp[3][5] = T`

Code:

```
let canPartitionBottomUp = function (num) {
    const n = num.length;
    // find the total sum
    let sum = 0;
    for (let i = 0; i < n; i++) sum += num[i];

    // if sum is an odd number we can't have two subsets with equal sum
    if (sum % 2 !== 0) return false;

    // we are trying to find a subset of given numbers that has a total of
'sum/2'
    sum /= 2;

    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]) {
                dp[i][s] = dp[i - 1][s - num[i]];
            }
        }
    }

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

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

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