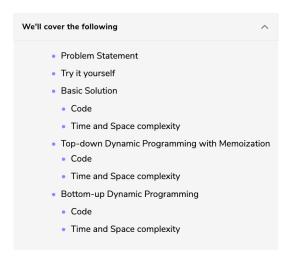




Equal Subset Sum Partition (medium)



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 subsets is 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} & {2, 3}
```

Example 2:

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

Example 3:

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

Try it yourself

This problem looks similar to the 0/1 Knapsack problem. Try solving it before moving on to see the solution:

Basic Solution

This 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 has an equal sum.

Assume that 5 represents the total sum of all the given numbers. Then the two equal subsets must have a sum equal to 5/2. This essentially transforms our problem to: "Find a subset of the given numbers that has a total sum of 5/2."

So our brute-force algorithm will look like:

```
1 for each number 'i'
2 create a new set which INCLUDES number 'i' if it does not exceed 'S/2', and recursively
3 | process the remaining numbers
4 create a new set WITHOUT number 'i', and recursively process the remaining items
5 return true if any of the above sets has a sum equal to 'S/2', otherwise return false
```

Code

Here is the code for the brute-force solution:

```
Pvthon3
                          ⊙ C++
                                       ic IS
     class PartitionSet {
      public boolean canPartition(int[] num) {
        int sum = 0;
for (int i = 0; i < num.length; i++)</pre>
          sum += num[i];
         if(sum % 2 != 0)
         return this.canPartitionRecursive(num, sum/2, 0);
      private boolean canPartitionRecursive(int[] num, int sum, int currentIndex) {
         if (sum == 0)
         if(num.length == 0 || currentIndex >= num.length)
           return false;
         // if the number at currentIndex exceeds the sum, we shouldn't process this if( num[currentIndex] \Leftarrow sum ) {
           if(canPartitionRecursive(num, sum - num[currentIndex], currentIndex + 1))
             return true:
Run
                                                                                                        Reset []
```

Time and Space complexity

The time complexity of the above algorithm is exponential $O(2^n)$, where 'n' represents the total number. The space complexity is O(n), 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. As stated in previous lessons, memoization is when we store the results of all the previously solved sub-problems so we can return the results from memory if we encounter a problem that has already been solved.

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 <code>canPartitionRecursive()</code>.

Code

Here is the code for Top-down DP with memoization:

```
1 class PartitionSet {
2
3    public boolean canPartition(int[] num) {
4    int sum = 0;
5    for (int i = 0; i < num.length; i++)
6    sum += num[i];
7
8    // if 'sum' is a an odd number, we can't have two subsets with equal sum
9    if (sum % 2 != 0)
10        return false;
11
12    Boolean[][] dp = new Boolean[num.length][sum / 2 + 1];
13    return this.canPartitionRecursive(dp, num, sum / 2, 0);
14    }
15
16    private boolean canPartitionRecursive(Boolean[][] dp, int[] num, int sum, int currentIndex) {
17    // base check</pre>
```

```
if (sum == 0)
    return true;

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

return false;

// if we have not already processed a similar problem

if (dp[currentIndex][sum] == null) {
    // 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) {
    Run</pre>

Run
```

Time and Space complexity

The above algorithm has the 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

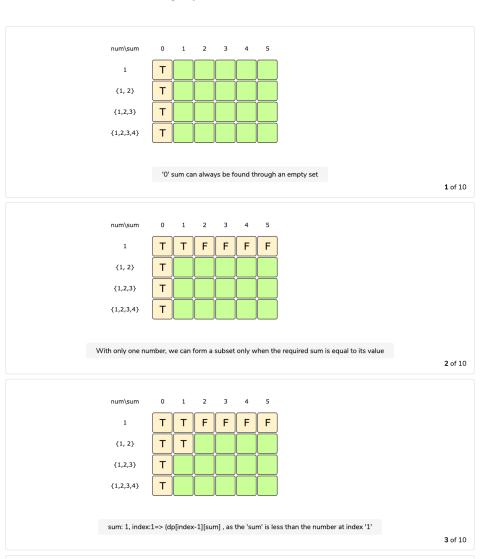
Let's try to populate our dp[][] array from the above solution by working in a bottom-up fashion.
Essentially, we want to find if we can make all possible sums with every subset. This means, dp[i][s] will be 'true' if we can make the sum 's' from the first 'i' numbers.

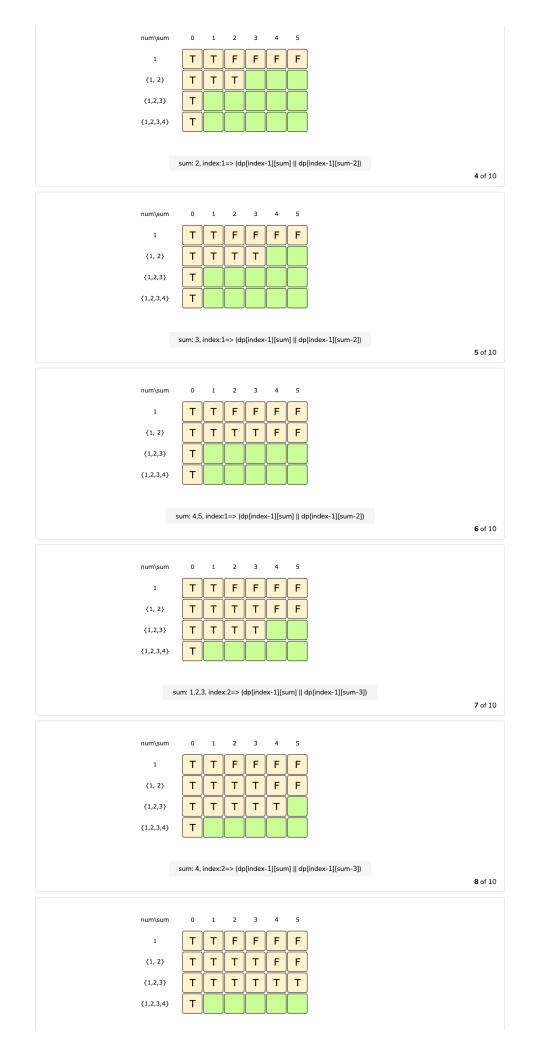
So, for each number at index 'i' ($0 \le i \le num.length$) and sum 's' ($0 \le s \le S/2$), we have two options:

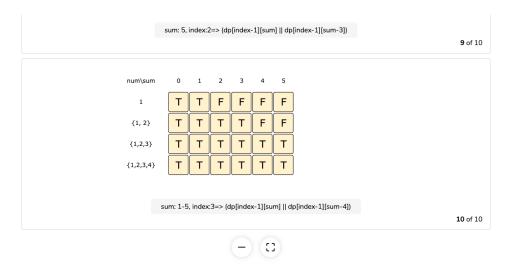
- 1. Exclude the number. In this case, we will see if we can get 's' from the subset excluding this number: dp[i-1][s]
- 2. Include the number if its value is not more than 's'. In this case, we will see if we can find a subset to get the remaining sum: dp[i-1][s-num[i]]

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:







From the above visualization, we can clearly see that it is possible to partition the given set into two subsets with equal sums, as shown by bottom-right cell: $dp[3][5] \Rightarrow T$

Code

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

```
class PartitionSet {

public boolean canPartition(int[] num) {

int n = num.length;

// find the total sum

int sum = 0;

for (int i = 0; i < n; i++)

sum += num[i];

// if 'sum' is a an odd number, we can't have two subsets with same total

if (sum % 2 != 0)

return false;

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

sum /= 2;

boolean[][] dp = new boolean[n][sum + 1];

// populate the sum=0 columns, as we can always for '0' sum with an empty set

for(int 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(int s=1; s <= sum; s++) {

dp[0][s] = (num[0] == s ? true : false);
}

// process all subsets for all sums

Run

Save Reset C:
```

Time and Space complexity

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

