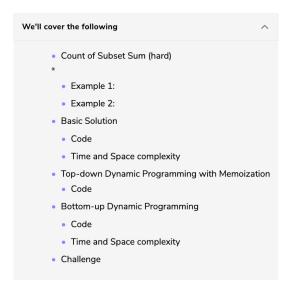




Solution Review: Problem Challenge 1



Count of Subset Sum (hard)

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

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

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

Here is the code for the brute-force solution:

```
1 class SubsetSum {
2
3 public int countSubsets(int[] num, int sum) {
4 return this.countSubsetsRecursive(num, sum, 0);
5 }
6
7 private int countSubsetsRecursive(int[] num, int sum, int currentIndex) {
8  // base checks
9  if (sum == 0)
10  return 1;
11
12  if(num.length == 0 || currentIndex >= num.length)
13 return 0;
```

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

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

return sum1 + sum2;
}

public static void main(String[] args) {
SubsetSum ss = new SubsetSum();

Run

Save

Reset C:
```

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), 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

Here is the code:

```
Python3
                  G C++
public int countSubsets(int[] num, int sum) {
  Integer[][] dp = new Integer[num.length][sum + 1];
  return this.countSubsetsRecursive(dp, num, sum, 0);
private int countSubsetsRecursive(Integer[][] dp, int[] num, int sum, int currentIndex) {
 if (sum == 0)
   return 1:
  if(num.length == 0 || currentIndex >= num.length)
    return 0;
  if(dp[currentIndex][sum] == null) {
    int sum1 = 0:
    if( num[currentIndex] <= sum )</pre>
     sum1 = countSubsetsRecursive(dp, num, sum - num[currentIndex], currentIndex + 1);
    int sum2 = countSubsetsRecursive(dp, num, sum, currentIndex + 1);
    dp[currentIndex][sum] = sum1 + sum2;
                                                                                 Save Reset []
```

Bottom-up Dynamic Programming

We will try to find if we can make all possible sums with every subset to populate the array db [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]]

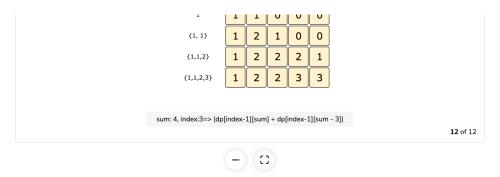
To find the total sets, we will add both of the above two values:

```
dp[index][sum] = dp[index-1][sum] + dp[index-1][sum-num[index]])
```

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







Code

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

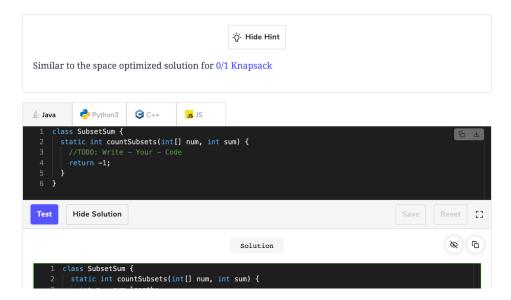
```
Python3
                              G C++
                                             JS JS
       lass SubsetSum {
       public int countSubsets(int[] num, int sum) {
         int n = num.length;
int[][] dp = new int[n][sum + 1];
          // populate the sum=0 columns, as we will always have an empty set for zero sum
for(int i=0; i < n; i++)</pre>
            dp[i][0] = 1;
          for(int s=1; s <= sum ; s++) {
    dp[0][s] = (num[0] == s ? 1 : 0);
          // process all subsets for all sums
for(int i=1; i < num.length; i++) {</pre>
            for(int s=1; s <= sum; s++) {
              dp[i][s] = dp[i-1][s];
               if(s >= num[i])
                 dp[i][s] += dp[i-1][s-num[i]];
          return dp[num.length-1][sum];
Run
                                                                                                              Save Reset []
```

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 desired sum.

Challenge

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



```
int n = num.tengtn;
int[] dp = new int[sum + 1];
dp[0] = 1;

// 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[s] = (num[0] == s ? 1 : 0);
}

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

return dp[sum];
}

return dp[sum];
}
```



Next \rightarrow

Problem Challenge 1

Problem Challenge 2



! Report an Issue ? Ask a Question