

Solution Review: Problem Challenge 1

We'll cover the following

- Count of Subset Sum (hard)
 - *
 - Example 1:
 - Example 2:
 - Basic Solution
 - Code
 - Time and Space complexity
 - Top-down Dynamic Programming with Memoization
 - Code
 - Bottom-up Dynamic Programming
 - Code
 - Time and Space complexity
 - Challenge

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:

```
1 for each number 'i'
2   create a new set which includes number 'i' if it does not exceed 'S', and recursively
3   process the remaining numbers and sum
4   create a new set without number 'i', and recursively process the remaining numbers
5 return the count of subsets who has a sum equal to 'S'
```

Code

Here is the code for the brute-force solution:

```
Java Python3 C++ JS
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;
```

```

13     return 0;
14
15     // recursive call after selecting the number at the currentIndex
16     // if the number at currentIndex exceeds the sum, we shouldn't process this
17     int sum1 = 0;
18     if( num[currentIndex] <= sum )
19         sum1 = countSubsetsRecursive(num, sum - num[currentIndex], currentIndex + 1);
20
21     // recursive call after excluding the number at the currentIndex
22     int sum2 = countSubsetsRecursive(num, sum, currentIndex + 1);
23
24     return sum1 + sum2;
25 }
26
27 public static void main(String[] args) {
28     SubsetSum ss = new SubsetSum();

```

Run Save 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 $O(n)$, 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:

Java Python3 C++ JS

```

1 class SubsetSum {
2
3     public int countSubsets(int[] num, int sum) {
4         Integer[][] dp = new Integer[num.length][sum + 1];
5         return this.countSubsetsRecursive(dp, num, sum, 0);
6     }
7
8     private int countSubsetsRecursive(Integer[][] dp, int[] num, int sum, int currentIndex) {
9         // base checks
10        if (sum == 0)
11            return 1;
12
13        if(num.length == 0 || currentIndex >= num.length)
14            return 0;
15
16        // check if we have not already processed a similar problem
17        if(dp[currentIndex][sum] == null) {
18            // recursive call after choosing the number at the currentIndex
19            // if the number at currentIndex exceeds the sum, we shouldn't process this
20            int sum1 = 0;
21            if( num[currentIndex] <= sum )
22                sum1 = countSubsetsRecursive(dp, num, sum - num[currentIndex], currentIndex + 1);
23
24            // recursive call after excluding the number at the currentIndex
25            int sum2 = countSubsetsRecursive(dp, num, sum, currentIndex + 1);
26
27            dp[currentIndex][sum] = sum1 + sum2;
28        }

```

Run 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

`dp[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:

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

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

1 of 12

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

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

2 of 12

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

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

3 of 12

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

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

4 of 12

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

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

5 of 12

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

{1,1,2,3} 1

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

6 of 12

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

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

7 of 12

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

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

8 of 12

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

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

9 of 12

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

sum: 1,2, index:3=> dp[index-1][sum] , as the sum is less than the element at index '3'

10 of 12

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

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

11 of 12

num\sum	0	1	2	3	4
1	1	1	0	0	0

	1	1	0	0	0
{1, 1}	1	2	1	0	0
{1,1,2}	1	2	2	2	1
{1,1,2,3}	1	2	2	3	3

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

12 of 12

Code

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

```

1 class SubsetSum {
2
3     public int countSubsets(int[] num, int sum) {
4         int n = num.length;
5         int[][] dp = new int[n][sum + 1];
6
7         // populate the sum=0 columns, as we will always have an empty set for zero sum
8         for(int i=0; i < n; i++)
9             dp[i][0] = 1;
10
11         // with only one number, we can form a subset only when the required sum is equal to its value
12         for(int s=1; s <= sum; s++) {
13             dp[0][s] = (num[0] == s ? 1 : 0);
14         }
15
16         // process all subsets for all sums
17         for(int i=1; i < num.length; i++) {
18             for(int s=1; s <= sum; s++) {
19                 // exclude the number
20                 dp[i][s] = dp[i-1][s];
21                 // include the number, if it does not exceed the sum
22                 if(s >= num[i])
23                     dp[i][s] += dp[i-1][s-num[i]];
24             }
25         }
26
27         // the bottom-right corner will have our answer.
28         return dp[num.length-1][sum];
29     }
30 }

```

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?

Hide Hint

Similar to the space optimized solution for [0/1 Knapsack](#)

```

1 class SubsetSum {
2     static int countSubsets(int[] num, int sum) {
3         //TODO: Write - Your - Code
4         return -1;
5     }
6 }

```

Test Hide Solution Save Reset

Solution

```

1 class SubsetSum {
2     static int countSubsets(int[] num, int sum) {
3         //TODO: Write - Your - Code
4         return -1;
5     }
6 }

```

```
3     int n = num.length;
4     int[] dp = new int[sum + 1];
5     dp[0] = 1;
6
7     // with only one number, we can form a subset only when the required sum is equal to its value
8     for(int s=1; s <= sum ; s++) {
9         dp[s] = (num[0] == s ? 1 : 0);
10    }
11
12    // process all subsets for all sums
13    for(int i=1; i < num.length; i++) {
14        for(int s=sum; s >= 0; s--) {
15            if(s >= num[i])
16                dp[s] += dp[s-num[i]];
17        }
18    }
19
20    return dp[sum];
21 }
22 }
23
```



[← Back](#)

Problem Challenge 1

[Next →](#)

Problem Challenge 2

☒ Mark as Completed

 Report an Issue  Ask a Question