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 (https://www.educative.io/collection/page/5668639101419520/5671464854355968/612696812473 5488/). The only difference in this problem is that we need to count the number of subsets, whereas in Subset Sum

(https://www.educative.io/collection/page/5668639101419520/5671464854355968/612696812473





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 JS
    def count_subsets(num, sum):
 2
       return count_subsets_recursive(num, sum, 0)
 3
 4
 5
    def count_subsets_recursive(num, sum, currentIndex):
 6
      # base checks
 7
      if sum == 0:
 8
        return 1
 9
      n = len(num)
10
      if n == 0 or currentIndex >= n:
        return 0
11
12
      # recursive call after selecting the number at the currentIndex
13
14
      # if the number at currentIndex exceeds the sum, we shouldn't process this
15
      sum1 = 0
16
      if num[currentIndex] <= sum:</pre>
17
        sum1 = count_subsets_recursive(
18
          num, sum - num[currentIndex], currentIndex + 1)
19
20
      # recursive call after excluding the number at the currentIndex
21
      sum2 = count_subsets_recursive(num, sum, currentIndex + 1)
22
23
      return sum1 + sum2
24
25
26 def main():
      print("Total number of subsets " + str(count_subsets([1, 1, 2, 3], 4)))
27
28
       print("Total number of subsets: " + str(count subsets([1, 2, 7, 1, 5], 9)))
\triangleright
                                                                                   8
```

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 twodimensional 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.



Here is the code:

```
👙 Java
            Python3
                          ⊘ C++
                                       JS JS
 1 def count subsets(num, sum):
      # create a two dimensional array for Memoization, each element is initialized to '-1^{\circ}
 3
       dp = [[-1 \text{ for } x \text{ in } range(sum+1)] \text{ for } y \text{ in } range(len(num))]
       return count_subsets_recursive(dp, num, sum, 0)
 5
 6
 7
    def count_subsets_recursive(dp, num, sum, currentIndex):
 8
      # base checks
 9
      if sum == 0:
10
         return 1
11
12
      n = len(num)
13
      if n == 0 or currentIndex >= n:
14
         return 0
15
16
      # check if we have not already processed a similar problem
17
       if dp[currentIndex][sum] == −1:
         # recursive call after choosing the number at the currentIndex
18
19
         # if the number at currentIndex exceeds the sum, we shouldn't process this
20
         sum1 = 0
21
         if num[currentIndex] <= sum:</pre>
22
           sum1 = count subsets recursive(
23
             dp, num, sum - num[currentIndex], currentIndex + 1)
24
25
         # recursive call after excluding the number at the currentIndex
26
         sum2 = count_subsets_recursive(dp, num, sum, currentIndex + 1)
27
28
         dp[currentIndex][sum] = sum1 + sum2
\triangleright
                                                                                             \leftarrow
```

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:

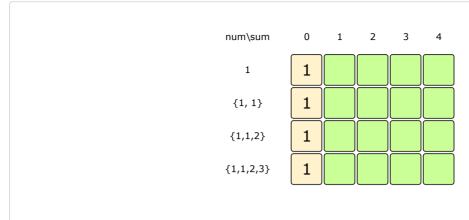
- 1. Exclude the number. Count all the subsets without the given number up to the given sum=> dp[index-1][sum]
- 2. Include the number if its value is not more than the 'sum'. In this case, we will count all the subsets to get the remaining sum => dp[index-1][sum-num[index]]

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]])
```

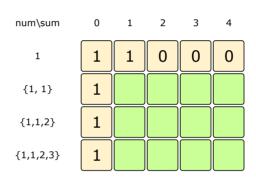
Let's start with our base case of size zero:





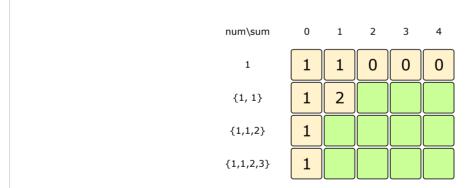
'0' sum can always be found through an empty set

1 of 12



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

2 of 12



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				



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

7 of 12



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

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

of 12

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

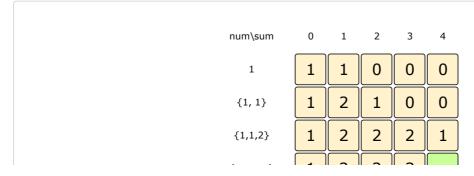
sum: 4, index:2=> (dp[index-1][sum] + dp[index-1][sum - 2])

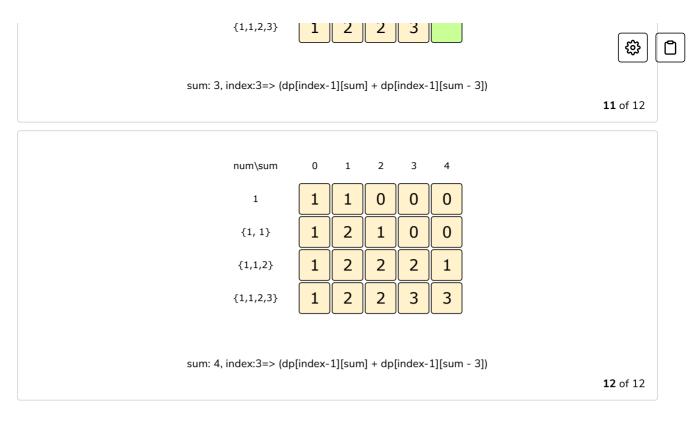
of 12

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

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

of 12

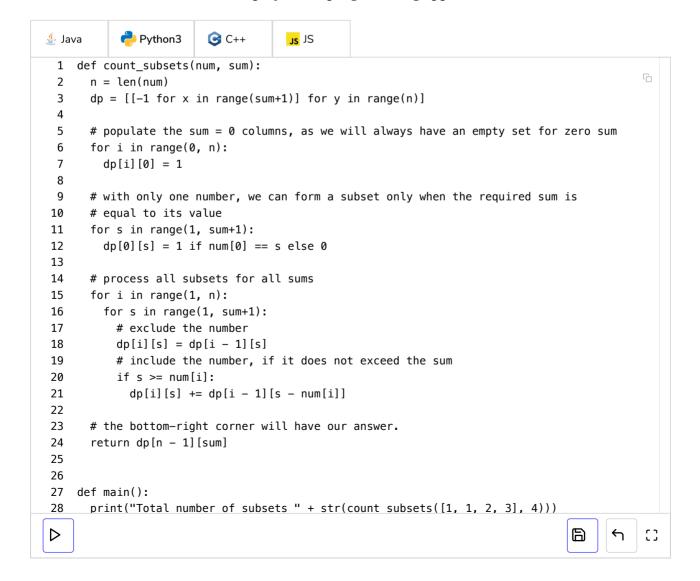




- []

Code

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



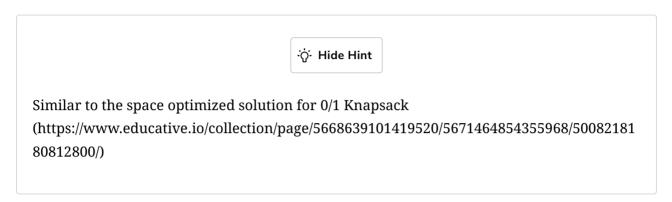




The above solution has the time and space complexity of O(N*S), where 'N' represents total numbers and 'S' is the desired sum.

Challenge

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



```
Python3
                           G C++
                                        Js JS
👙 Java
 1 def count_subsets(num, sum):
       n = len(num)
 3
       dp = [0 \text{ for } x \text{ in range(sum+1)}]
 4
       dp[0] = 1
 5
 6
       # with only one number, we can form a subset only when the required sum is equal to the
 7
       for s in range(1, sum+1):
 8
         dp[s] = 1 \text{ if } num[0] == s \text{ else } 0
 9
10
       # process all subsets for all sums
11
       for i in range(1, n):
12
        for s in range(sum, -1, -1):
13
           if s >= num[i]:
14
             dp[s] += dp[s - num[i]]
15
       return dp[sum]
16
17
18
19 def main():
       print("Total number of subsets " + str(count_subsets([1, 1, 2, 3], 4)))
20
21
       print("Total number of subsets: " + str(count_subsets([1, 2, 7, 1, 5], 9)))
22
23
24 main()
25
26
27
28
                                                                                                    []
\triangleright
                                                                                       \leftarrow
```





Problem Challenge 1



? Ask a Question

 $(https://discuss.educative.io/tag/solution-review-problem-challenge-1_pattern--01-knapsack-dynamic-programming_grokking-the-coding-interview-patterns-for-coding-questions)\\$