

Count of Subset Sum

We'll cover the following



- Problem Statement
 - *
 - Example 1:
 - Example 2:
- Basic Solution
 - Code
- Top-down Dynamic Programming with Memoization
 - Code
- Bottom-up Dynamic Programming
 - Code
- Challenge

Problem Statement

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/5668639101419520/5633779737559040/5646239437684736>). The only difference in this problem is that we need to count the number of subsets, whereas in the Subset Sum (<https://www.educative.io/collection/page/5668639101419520/5633779737559040/5646239437684736>) we only wanted to know if there exists a subset with the given sum.

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

JS

Python3

C++

```
1 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:
11        return 0
12
13    # recursive call after selecting the number at the currentIndex
14    # if the number at currentIndex exceeds the sum, we shouldn't process this
15    sum1 = 0
16    if num[currentIndex] <= sum:
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():
27     print("Total number of subsets " + str(count_subsets([1, 1, 2, 3], 4)))
28     print("Total number of subsets: " + str(count_subsets([1, 2, 7, 1, 5], 9)))
29
30
31 main()
```



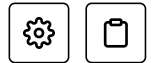
The time complexity of the above algorithm is exponential $O(2^n)$, where 'n' represents the total number. The space complexity is $O(n)$, this memory is used to store the recursion stack.

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

JS

Python3

C++

```
1 def count_subsets(num, sum):
2     # create a two dimensional array for Memoization, each element is initialized to '-1'
3     dp = [[-1 for x in range(sum+1)] for y in range(len(num))]
4     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:
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        sum1 = 0
21        if num[currentIndex] <= sum:
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
29
30    return dp[currentIndex][sum]
31
32
33 def main():
34     print("Total number of subsets " + str(count_subsets([1, 1, 2, 3], 4)))
35     print("Total number of subsets: " + str(count_subsets([1, 2, 7, 1, 5], 9)))
36
37
38 main()
```



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				

$$\text{sum: 1, index:1} \Rightarrow (dp[index-1][sum] + dp[index-1][sum - 1])$$

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				

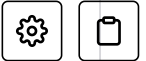
sum: 3, index:2=> (dp[index-1][sum] + dp[index-1][sum - 2])

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}	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:

Java

JS

Python3

C++

```
1 def count_subsets(num, sum):
2     n = len(num)
```



```

3  dp = [[-1 for x in range(sum+1)] for y in range(n)]
4
5  # populate the sum = 0 columns, as we will always have an empty set for zero sum
6  for i in range(0, n):
7      dp[i][0] = 1
8
9  # with only one number, we can form a subset only when the required sum is
10 # equal to its value
11 for s in range(1, sum+1):
12     dp[0][s] = 1 if num[0] == s else 0
13
14 # process all subsets for all sums
15 for i in range(1, n):
16     for s in range(1, sum+1):
17         # exclude the number
18         dp[i][s] = dp[i - 1][s]
19         # include the number, if it does not exceed the sum
20         if s >= num[i]:
21             dp[i][s] += dp[i - 1][s - num[i]]
22
23 # the bottom-right corner will have our answer.
24 return dp[n - 1][sum]
25
26
27 def main():
28     print("Total number of subsets " + str(count_subsets([1, 1, 2, 3], 4)))
29     print("Total number of subsets: " + str(count_subsets([1, 2, 7, 1, 5], 9)))
30
31
32 main()

```



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

Challenge

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

Show Hint

Java

JS

Python3

C++

```

1  def count_subsets(num, sum):
2      # TODO: Write - Your - Code
3      return -1
4

```



Solution



```

1  def count_subsets(num, sum):
2      n = len(num)
3      dp = [0 for x 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 number
7     for s in range(1, sum+1):
8         dp[s] = 1 if num[0] == s 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
16    return dp[sum]
17
18
19 def main():
20     print("Total number of subsets " + str(count_subsets([1, 1, 2, 3], 4)))
21     print("Total number of subsets: " + str(count_subsets([1, 2, 7, 1, 5], 9)))
22
23
24 main()

```



← Back

Next →

Minimum Subset Sum Difference

Target Sum

☒ Mark as Completed



Report an Issue



Ask a Question

(https://discuss.educative.io/tag/count-of-subset-sum__pattern-1-01-knapsack__grokking-dynamic-programming-patterns-for-coding-interviews)