Solution Review: Problem Challenge 1

We'll cover the following

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:

Java | Python3 | C++ | JS

```python
def count_subsets(num, sum):
  return count_subsets_recursive(num, sum, 0)


def count_subsets_recursive(num, sum, currentIndex):
  # base checks
  if sum == 0:
    return 1
  n = len(num)
  if n == 0 or currentIndex >= n:
    return 0

  # recursive call after selecting the number at the currentIndex
  # if the number at currentIndex exceeds the sum, we shouldn't process this
  sum1 = 0
  if num[currentIndex] <= sum:
    sum1 = count_subsets_recursive(
      num, sum - num[currentIndex], currentIndex + 1)

  # recursive call after excluding the number at the currentIndex
  sum2 = count_subsets_recursive(num, sum, currentIndex + 1)

  return sum1 + sum2


def main():
  print("Total number of subsets " + str(count_subsets([1, 1, 2, 3], 4)))
  print("Total number of subsets: " + str(count_subsets([1, 2, 7, 1, 5], 9)))


main()
```

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

```python
def count_subsets(num, sum):
  # create a two dimensional array for Memoization, each element is initialized to '-1'
  dp = [[-1 for x in range(sum+1)] for y in range(len(num))]
  return count_subsets_recursive(dp, num, sum, 0)


def count_subsets_recursive(dp, num, sum, currentIndex):
  # base checks
  if sum == 0:
    return 1

  n = len(num)
  if n == 0 or currentIndex >= n:
    return 0

  # check if we have not already processed a similar problem
  if dp[currentIndex][sum] == -1:
    # recursive call after choosing the number at the currentIndex
    # if the number at currentIndex exceeds the sum, we shouldn't process this
    sum1 = 0
    if num[currentIndex] <= sum:
      sum1 = count_subsets_recursive(
        dp, num, sum - num[currentIndex], currentIndex + 1)
```

```python
    # recursive call after excluding the number at the currentIndex
    sum2 = count_subsets_recursive(dp, num, sum, currentIndex + 1)

    dp[currentIndex][sum] = sum1 + sum2

  return dp[currentIndex][sum]


def main():
  print("Total number of subsets " + str(count_subsets([1, 1, 2, 3], 4)))
  print("Total number of subsets: " + str(count_subsets([1, 2, 7, 1, 5], 9)))


main()
```

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 `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:

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])

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])

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'

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])

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])

Code

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

Java | Python3 | C++ | JS

```python
def count_subsets(num, sum):
  n = len(num)
  dp = [[-1 for x in range(sum+1)] for y in range(n)]

  # populate the sum = 0 columns, as we will always have an empty set for zero sum
  for i in range(0, n):
    dp[i][0] = 1

  # with only one number, we can form a subset only when the required sum is
  # equal to its value
  for s in range(1, sum+1):
    dp[0][s] = 1 if num[0] == s else 0

  # process all subsets for all sums
  for i in range(1, n):
    for s in range(1, sum+1):
      # exclude the number
      dp[i][s] = dp[i - 1][s]
      # include the number, if it does not exceed the sum
      if s >= num[i]:
        dp[i][s] += dp[i - 1][s - num[i]]

  # the bottom-right corner will have our answer.
  return dp[n - 1][sum]


def main():
  print("Total number of subsets " + str(count_subsets([1, 1, 2, 3], 4)))
  print("Total number of subsets: " + str(count_subsets([1, 2, 7, 1, 5], 9)))


main()
```

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

Java | Python3 | C++ | JS

```python
def count_subsets(num, sum):
  n = len(num)
  dp = [0 for x in range(sum+1)]
  dp[0] = 1

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

```python
  for s in range(1, sum+1):
    dp[s] = 1 if num[0] == s else 0

  # process all subsets for all sums
  for i in range(1, n):
    for s in range(sum, -1, -1):
      if s >= num[i]:
        dp[s] += dp[s - num[i]]

  return dp[sum]


def main():
  print("Total number of subsets " + str(count_subsets([1, 1, 2, 3], 4)))
  print("Total number of subsets: " + str(count_subsets([1, 2, 7, 1, 5], 9)))


main()
```

Run Save Reset

← Back
Problem Challenge 1

Next →
Problem Challenge 2

Completed

Report an Issue Ask a Question