Annamariya Tharayil

Following 68 Followers About

Target Sum — Day 48(Python)

 Annamariya Tharayil Dec 22, 2020 · 5 min read

Photo by Immo Wegmann on Unsplash

Today's problem is a variation of the Knapsack problem. The question is from leetcode and a "Medium" tagged question. Let us jump right into the question.

494. Target Sum

You are given a list of non-negative integers, a1, a2, …, an, and a target, S. Now you have 2 symbols `+` and `-`. For each integer, you should choose one from `+` and `-` as its new symbol.

Find out how many ways to assign symbols to make the sum of integers equal to target S.

Example 1:

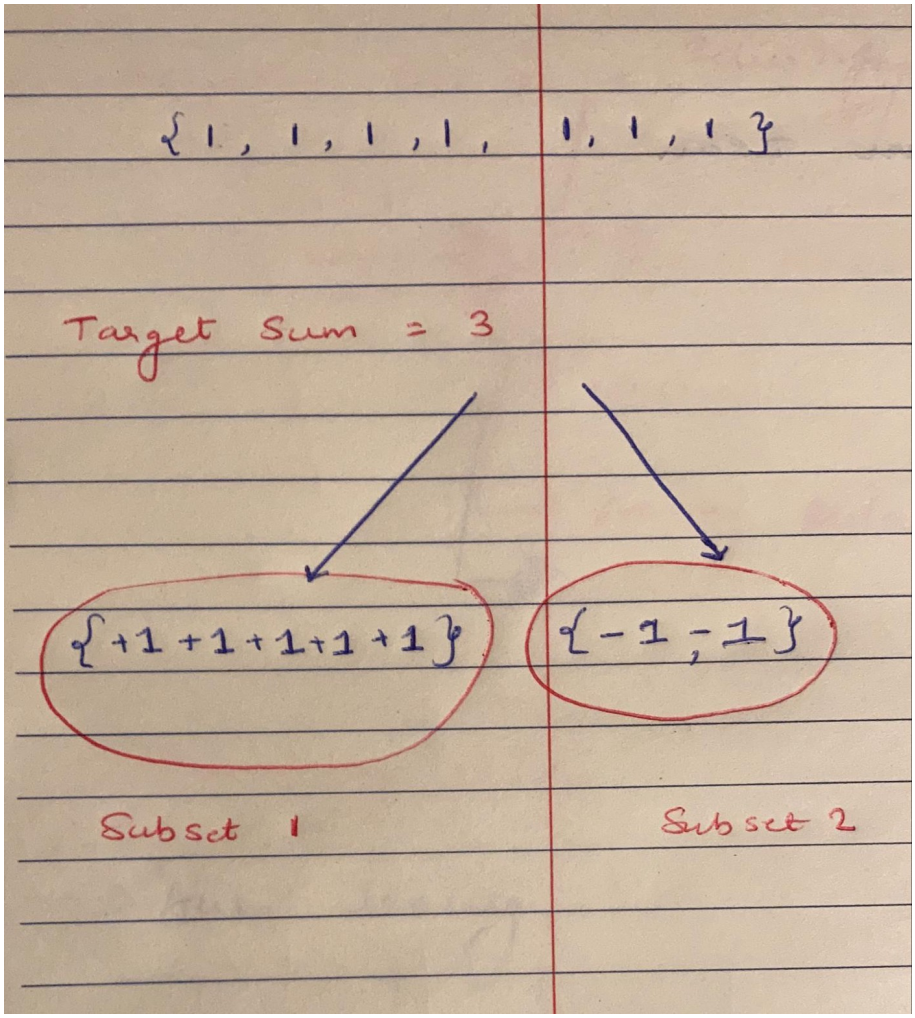
```
Input: nums is [1, 1, 1, 1, 1], S is 3.
Output: 5
Explanation:

-1+1+1+1+1 = 3
+1-1+1+1+1 = 3
+1+1-1+1+1 = 3
```

```
There are 5 ways to assign symbols to make the sum of nums be target
3.
```

Constraints:

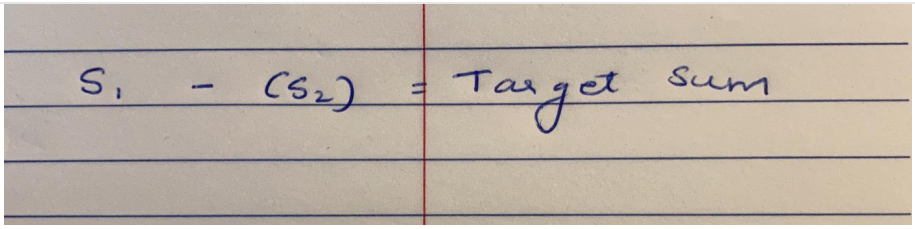
- The length of the given array is positive and will not exceed 20.
- The sum of elements in the given array will not exceed 1000.
- Your output answer is guaranteed to be fitted in a 32-bit integer.

The problem states that we need to count the ways, such that when we assign signs (+ or -) to the number in the list, they reach a target number. Some numbers would be positive, while some numbers will be negative. Why don't we consider positive numbers as one set and negative numbers as one another set?



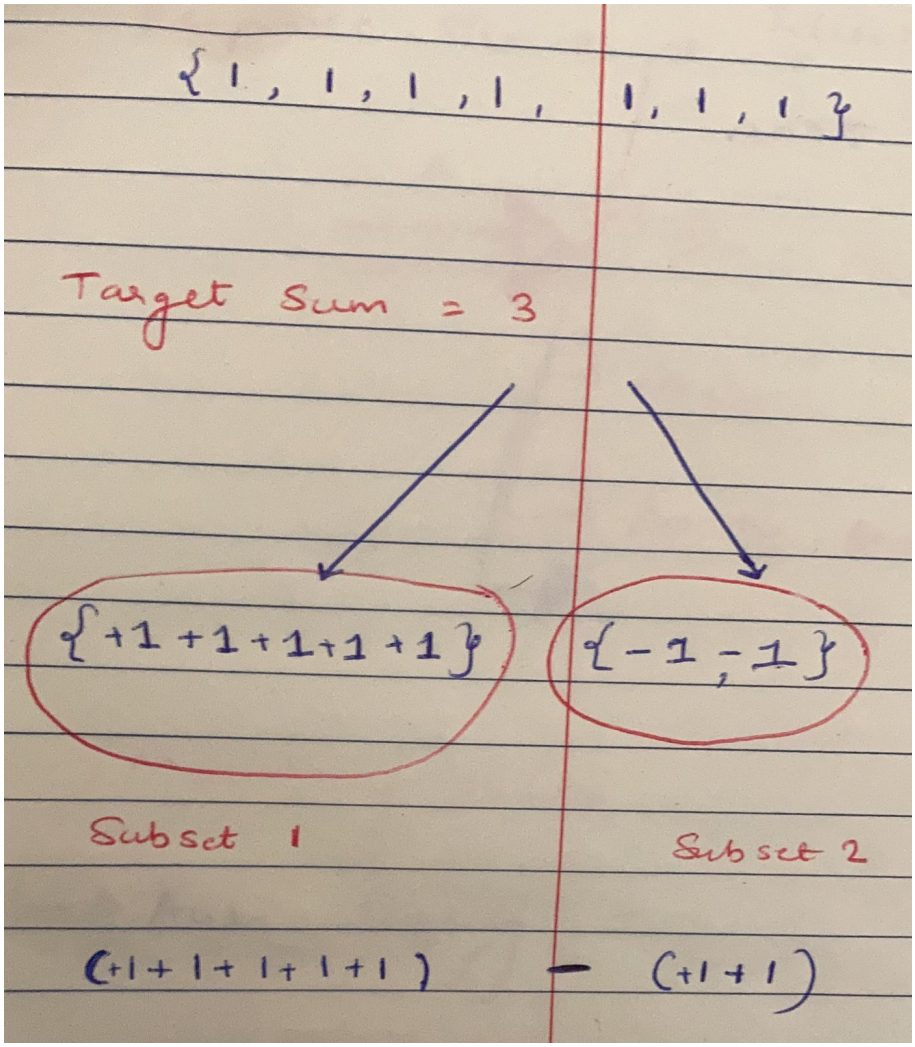
Dividing into two subsets

Primarily, we need to find the number of subsets that reach a target sum. How do we find that target sum? Let us look into some Math.



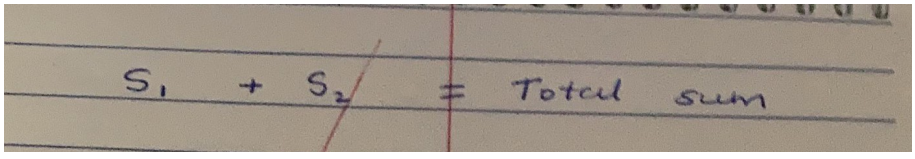
In the second equation, why did we perform a subtraction? Aren't we supposed to perform addition? If we take the common sign i.e. (-) outside, we will get a set of numbers to add. Okay, this is complicated.

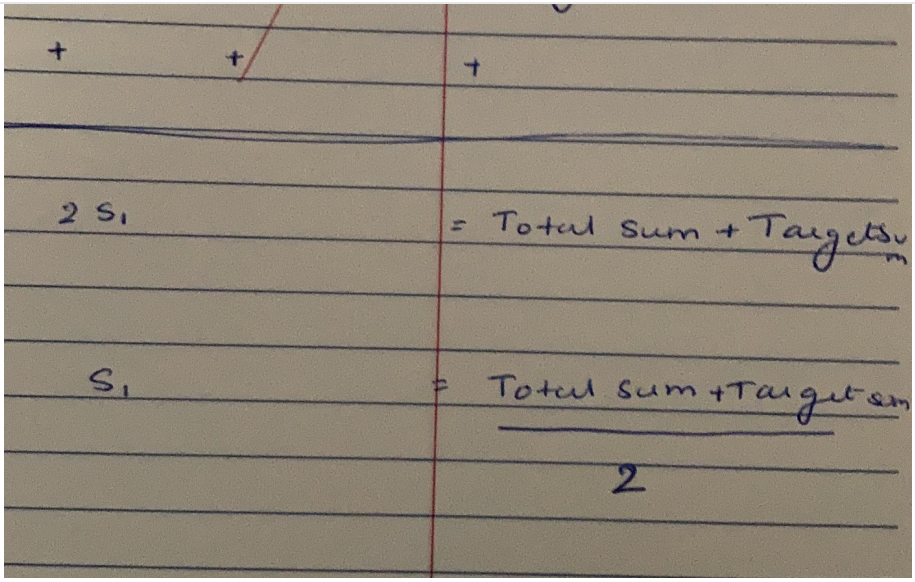
Let us take an example.



Hopefully, the above example clears the point.

We need to find the number of subsets that reach a particular target value. How do we know the target value? Some more Math follows!





We need to find the number of subsets that total up to the above calculation. We can use the Knapsack algorithm to find the number of subsets except with minor modifications.

Let us first write the code for the preprocessing steps. The preprocessing step is finding the required sum for the subset to be found. We calculated the formula to find the sum of the subset in the above picture.

```
class TargetSum:
    def findTargetSumWays(self, nums: List[int], S: int) -> int:
        total_sum = sum(nums)
        required_sum = (total_sum + S) // 2
```

Let us move to the crux of solving this problem. We know the required total value the subset should have, now we need to find the number of subsets that can be formed with its sum as the required sum.

We have a few numbers in our list, and each number will have two choices, either to include in subset 1 or subset 2. For now, we are taking into consideration if we want to include the number in subset 1. The problem is similar to the Knapsack problem where each item had 2 choices, either to be included or excluded in our knapsack.

We will be directly moving to the tabulated method of finding the solution since we already know the recursive and memoized way of solving the problem.

The columns represent values from zero until the required sum, and the rows represent the number of elements in our input list.

When the required sum is 0, and there is no number in our list, we have a solution i.e. empty subset.

Next, we will take elements from our list, check if it is lesser than the required sum, if yes, we have 2 choices.

1. We include it in our subset.
2. We exclude from our subset.

The final result is the value at the last row and column in our table.

Let us look into the code snippet.

```
class TargetSum:
  def findTargetSumWays(self, nums: List[int], S: int) -> int:
     total_sum = sum(nums)
     if total_sum < S or (total_sum + S)%2 != 0 :
        return 0
     required_sum = (total_sum + S) // 2
     memo = [[0 for j in range(required_sum+1)]for i in
range(len(nums)+1)]
     memo[0][0] = 1
     for i in range(1, len(memo)):
       for j in range(len(memo[0])):
         if nums[i-1] <= j:
            memo[i][j] = memo[i-1][j-nums[i-1]] + memo[i-1][j]
         else:
            memo[i][j] = memo[i-1][j]
     return memo[-1][-1]
```

In the leetcode question, our array can contain 0 as a number too. Hence we are not initializing the entire 0th index column with 1. Another edge case to consider if the given target is greater than the total sum of the array. We would return 0 as we will not find a solution if the target sum is greater than the total sum of the array.

Complexity analysis.

Time Complexity

We are creating a 2D array of size ((N)*(R)) and traversing through the entire table once. Hence the time complexity is ((N)*(R)), where N is the count of numbers in the array and R is the required sum.

Space Complexity.

We are creating a 2D array of size ((N)*(R)), hence the space complexity is ((N)*(R)), where N is the count of numbers in the array and R is the required sum.

I have used Aditya Verma's video as a reference to solve this problem.

I would love to hear your feedback about my posts. Do let me know if you have any comments or feedback.

Dynamic Programming 365dayschallenge Python Coding Arrays