494. Target Sum

Dynamic Programming

Problem Description

Medium

adding either a '+' (plus sign) or a '-' (minus sign) before each number in nums, then concatenate (join together) these numbers with their signs to form an expression. The end goal is to find out how many different expressions you can create that, once evaluated, equal the target.

You're given a list of numbers called nums and a single number called target. Your task is to form a mathematical expression by

For instance, if nums is [2, 1] and your target is 1, you could create the expression "+2-1", which equals 1. You want to find out all such possible expressions that result in the target.

Here's what you need to consider:

• You can't reorder the numbers – their order in the expression must match their order in the given nums list.

• The objective is to count the number of valid expressions, not necessarily to generate each one.

You can only use '+' or '-' before each number.

ntuition

Backtracking

Understanding the problem, we see that it resembles a classic problem in computer science known as the subset sum problem,

except it allows for both positive and negative summations. The crux is to figure out how to partition nums into two subsets where the difference between the sums of the subsets equals target.

obtain the target. This is basically the same as finding two subsets with a particular sum difference. Now, how do we solve this with dynamic programming? We can use a technique similar to the 0-1 Knapsack problem. The idea is to iteratively build up an array dp where each index represents a possible sum of numbers, starting from 0 up to some value that

First step is to notice a helpful fact: if we sum all numbers with a '+' and then subtract the sum of numbers with a '-', we should

depends on nums and target. Each cell in dp will tell us the number of ways to achieve that sum. But how do we compute the size of dp and its initial values? We define a new variable n which is the sum that we want one subset

to have so that the other subset has a sum of n + target (given that the total sum of nums is s). It turns out n should be (s target) / 2. We only proceed if s - target is even, otherwise, it's not possible to split nums into two such subsets. Once we have dp setup starting at dp[0] = 1 (there's one way to achieve a sum of 0: by choosing no numbers), we start

populating dp by iterating through each number in nums. For each v in nums, we iterate backwards through dp from n down to v

(since v is the smallest sum that including v can achieve) and update dp[j] to include the number of ways we can achieve the sum j - v. In the end, dp[n] gives us the number of different expressions we can form that equal target. The code provided implements this <u>dynamic programming</u> approach efficiently.

The provided solution uses a dynamic programming approach to solve the problem efficiently, much like the "0-1 Knapsack" problem. The solution involves some pre-processing steps and then iteratively filling out a dp array to count the number of ways to reach different sums.

• If the total sum s is less than target, it's not possible to create any expression equal to target.

∘ If s - target is odd, we can't split the array into two subsets with integer sums that have the needed difference.

Calculate the sum s of all numbers in nums. This is done to determine the sum of one subset (n) so that the difference with the other subset's sum will be target.

Here's a step-by-step breakdown of the implementation:

Check for two conditions that might make it impossible to create expressions equal to target:

equal the target.

Solution Approach

(by choosing no numbers). Iterate over each number v in nums. For each v, iterate backwards over the dp array from n down to v to avoid recomputing values that rely on the current index. This is crucial because we must not count any combination twice.

Initialize the dp array of size n + 1 with zeros and set dp [0] to 1 - this signifies that there is one way to create a sum of zero

Update the dp[j] value by adding the number of combinations that exist without the current number v (dp[j - v]). This step

accumulates the number of ways there are to achieve a sum of j by either including or excluding the current number v. By the end of the iteration, dp[-1] (which is dp[n]) will hold the number of possible expressions that can be created using nums to

Suppose nums = [1,2,3] and target = 1. We compute n = (s - target) / 2, which would be 2 in this case.

○ When v = 3, we update dp for j from 2 to 1, but this time there are no changes as 3 is too large to affect dp[1] or dp[2].

Example of dp Array Update Here's a hypothetical example to illustrate the dynamic programming update process:

Our dp array is [1, 0, 0, 0, 0] initially. During the iteration process:

The method described balances the need to consider both including and excluding each number in nums while ensuring each sum

is only counted once. It utilizes memory efficiently by only maintaining a one-dimensional array rather than a full matrix that

Example Walkthrough

we proceed.

The dp array is now [1, 1, 0].

The dp array is now [1, 1, 1].

For v = 2, update dp[j] from j = 2 to 2:

nums to sum up to target, which is "1 + 2 - 3".

to the target without redundantly computing combinations.

• When v = 1, we update dp to [1, 1, 0, 0, 0].

 \circ When v = 2, we update dp for j from 2 to 1, resulting in [1, 1, 1, 0, 0].

According to the solution approach, our first step is to calculate the sum s of all numbers in nums. Here, the sum is 1 + 2 + 3 = 6.

Next, we iterate over each number v in nums. For each v, we update the dp array from n down to v:

For v = 1, update dp[j] from j = 2 to 1. The dp array changes as follows:

 \circ dp[1] gets updated to dp[1] + dp[1 - 1] which is dp[1] = 0 + 1 = 1.

 \circ dp[2] gets updated to dp[2] + dp[2 - 2] which is dp[2] = 0 + 1 = 1.

def findTargetSumWays(self, nums: List[int], target: int) -> int:

if total_sum < target or (total_sum - target) % 2 != 0:</pre>

Calculate the sum of all numbers in the array

Check if the target is achievable or not

In the end, dp[n] gives us the count of the number of expressions that sum up to target.

would be required in a naive implementation of dynamic programming for this problem.

- Let's walk through a small example to illustrate the provided solution approach. Suppose we have nums = [1, 2, 3] and target = 2.
- Our next step is to determine the size of the dp array, which will help us count the number of ways to make up different sums. We calculate n = (s - target) / 2, which is (6 - 2) / 2 = 2. Now we initialize our dp array with zeros and set dp[0] to 1. So initially, our dp array is [1, 0, 0].

Then we check if the total sum s is less than target or if s - target is odd. In this case, 6 is not less than 2, and 6 - 2 is even, so

For v = 3, since our dp array size is only 3, we don't need to update it for v = 3 because 3 is larger than n. Finally, our dp array is [1, 1, 1], and the dp [-1] or dp [2] indicates that there is 1 possible expression that can be created using

This walkthrough demonstrates how the dynamic programming approach effectively counts the number of expressions equating

Solution Implementation

If the sum of nums is less than target, or the difference between sum and target

is not an even number, then return 0 because target can't be achieved

Initialize a list for dynamic programming, with a size of subset_sum + 1

There is always 1 way to achieve a sum of 0, which is by selecting no elements

For each number in nums, update the count of ways to achieve each sum <= subset_sum

Add the number of ways to achieve a sum of j before num was considered

Return the number of ways to achieve subset_sum, which indirectly gives us the number of ways

return 0 # Compute the subset sum we need to find to partition the array # into two subsets that give the desired target on applying + and - operations subset_sum = (total_sum - target) // 2

```
# Example usage:
```

Python

from typing import List

dp[0] = 1

for num in nums:

return dp[-1]

print(result) # Outputs: 5

solution = Solution()

total_sum = sum(nums)

 $dp = [0] * (subset_sum + 1)$

Update the dynamic programming table

dp[j] += dp[j - num]

result = solution.findTargetSumWays([1, 1, 1, 1, 1], 3)

for j in range(subset_sum, num - 1, -1):

to achieve the target sum using '+' and '-' operations

int sum = std::accumulate(nums.begin(), nums.end(), 0);

if (sum < target || (sum - target) % 2 != 0) return 0;</pre>

// Initialize the dynamic programming array with zeros

// Iterate over every number in the input array

for (int j = newTarget; j >= num; --j) {

int newTarget = (sum - target) / 2;

vector<int> dp(newTarget + 1, 0);

dp[j] += dp[j - num];

* Calculates the number of ways to assign '+' and '-'

let sum: number = nums.reduce((acc, val) => acc + val, 0);

* to make the sum of nums be equal to target.

for (int num : nums) {

return dp[newTarget];

};

/**

TypeScript

dp[0] = 1;

// Calculate the new target (n) which is the sum to be found

// If the sum is less than the target or the adjusted sum is odd, return 0

// and set dp[0] to 1 since there's one way to get sum 0: using no numbers

// This is to ensure we do not count any subset more than once

// Return the total number of ways to reach the 'newTarget', which corresponds

// to the number of ways to reach the original 'target' considering +/- signs

// If the sum is less than the target, or the difference is odd, there is no solution

// For each number, iterate backwards from the new target to the number's value

// Update the dp array to count additional ways to reach current sum (j)

// by adding the number of ways to reach the sum without the current number (j - num)

class Solution:

```
Java
class Solution {
    public int findTargetSumWays(int[] nums, int target) {
        // Initialize the sum of all numbers in nums
        int sum = 0;
        for (int num : nums) {
            sum += num;
       // If the sum is less than the target or (sum - target) is odd, it's not possible to partition
       if (sum < target || (sum - target) % 2 != 0) {</pre>
            return 0;
        // Compute the subset sum needed for one side of the partition
        int subsetSum = (sum - target) / 2;
       // Initialize a DP array to store the number of ways to reach a particular sum
        int[] dp = new int[subsetSum + 1];
       // There's one way to reach the sum of 0 — by not including any numbers
       dp[0] = 1;
       // Go through every number in nums
        for (int num : nums) {
            // Update the DP table from the end to the start to avoid overcounting
            for (int j = subsetSum; j >= num; j--) {
                // Increase the current dp value by the value from dp[j - num]
                dp[j] += dp[j - num];
        // Return the number of ways to reach the target sum
        return dp[subsetSum];
#include <vector>
#include <numeric> // For using accumulate function
class Solution {
public:
    int findTargetSumWays(vector<int>& nums, int target) {
       // Calculate the sum of all numbers in the vector
```

```
* @param {number[]} nums - Array of numbers to be used.
* @param {number} target - The target sum to be achieved.
* @return {number} - Number of ways to achieve the target sum.
*/
const findTargetSumWays = (nums: number[], target: number): number => {
   // Calculate the sum of the input array elements
```

```
if (sum < target || (sum - target) % 2 !== 0) {</pre>
          return 0;
      const totalNums: number = nums.length;
      // Calculate the subset sum that we need to find
      const subsetSum: number = (sum - target) / 2;
      // Initialize a DP array for storing number of ways to sum up to j with array elements
      let waysToSum: number[] = new Array(subsetSum + 1).fill(0);
      waysToSum[0] = 1; // Base case - there's one way to have a sum of zero (using no elements)
      // Fill the DP array
      for (let i = 1; i <= totalNums; ++i) {</pre>
          for (let j = subsetSum; j >= nums[i - 1]; ---j) {
              waysToSum[j] += waysToSum[j - nums[i - 1]]; // Update the ways to sum to j
      // Return the total ways to achieve the subset sum, which is equivalent to the target
      return waysToSum[subsetSum];
  };
  export { findTargetSumWays }; // Exporting the function to be used in other modules
from typing import List
class Solution:
   def findTargetSumWays(self, nums: List[int], target: int) -> int:
       # Calculate the sum of all numbers in the array
        total_sum = sum(nums)
       # Check if the target is achievable or not
       # If the sum of nums is less than target, or the difference between sum and target
       # is not an even number, then return 0 because target can't be achieved
       if total_sum < target or (total_sum - target) % 2 != 0:</pre>
            return 0
       # Compute the subset sum we need to find to partition the array
       # into two subsets that give the desired target on applying + and - operations
        subset_sum = (total_sum - target) // 2
```

Initialize a list for dynamic programming, with a size of subset_sum + 1

There is always 1 way to achieve a sum of 0, which is by selecting no elements

For each number in nums, update the count of ways to achieve each sum <= subset_sum

Add the number of ways to achieve a sum of j before num was considered

Return the number of ways to achieve subset_sum, which indirectly gives us the number of ways

```
Time and Space Complexity
```

result = solution.findTargetSumWays([1, 1, 1, 1, 1], 3)

 $dp = [0] * (subset_sum + 1)$

Update the dynamic programming table

dp[j] += dp[j - num]

for j in range(subset_sum, num - 1, -1):

to achieve the target sum using '+' and '-' operations

dp[0] = 1

for num in nums:

return dp[-1]

print(result) # Outputs: 5

Example usage:

solution = Solution()

The time complexity of the algorithm is 0(len(nums) * n), where len(nums) is the number of elements in the input list nums, and n is the computed value (sum(nums) - target) // 2. This is because the algorithm consists of a nested loop, where the outer loop runs for each element in nums, and the inner loop runs from n down to the value of the current element v.

The space complexity of the algorithm is 0(n + 1), which simplifies to 0(n), as there is a one-dimensional list dp of size n + 1elements used to store the intermediate results for the subsets that sum up to each possible value from 0 to n.