377. Combination Sum IV

Dynamic Programming

Problem Description

Medium Array

In this problem, you are given an array of unique integers nums and a target integer target. Your task is to find out how many different combinations of elements from nums can be added together to sum up to the target value. Importantly, combinations may use the same element from nums multiple times if needed.

For example, if nums = [1, 2, 3] and target = 4, there are seven combinations that you could use to get a sum of 4:

• 1+1+1+1 • 1+1+2

• 1+2+1 • 2+1+1 • 2+2 • 1+3 • 3+1

Intuition

The results must fit within a 32-bit integer range, meaning they should be between $-(2^31)$ and $(2^31)-1$.

The intuition for solving this problem comes from recognizing that it is similar to computing the number of ways to make change

for a certain amount of money given different coin denominations. Here, each integer in nums can be thought of as a coin denomination. To arrive at the solution, we apply <u>dynamic programming</u> because the problem asks for counting combinations, which suggests

overlapping subproblems and the need for memorization of previous results to optimize the computation. Each subproblem here is finding the number of ways to reach a smaller target, which we can use to build up the solution to a larger target.

The dp array is created where each index i represents the target sum, and the value at that index represents the number of ways to reach that sum using elements from nums. The base case is dp[0] = 1, indicating there's one way to reach a sum of 0, which is using no elements. We iterate through each sub-target from 1 to the desired target, and for each, we look at all elements in nums. If the current

element is less than or equal to the sub-target, we add to dp[i] the value of dp[i - x], where x is the value of the current element from nums. This signifies that all the ways to reach the sum i-x can contribute to ways to reach the sum i by adding x. By the end of the iterations, dp[target] contains the desired number of combinations.

The solution approach utilizes dynamic programming, a method for solving complex problems by breaking them down into simpler sub-problems. It is a helpful strategy when a problem has overlapping subproblems and optimal substructure, meaning the problem can be broken down into smaller, simpler subproblems which can be solved independently.

1 because there's exactly one combination to achieve a sum of 0: using no numbers from nums.

when adding x).

class Solution:

Solution Approach

The outer loop iterates through all sub-targets from 1 to target, and the inner loop goes through all the available numbers in nums. Whenever the current number x is less than or equal to the sub-target i, we update dp[i] by adding the number of ways to form the sum i-x. This is based on the assumption that if we have a way to arrive at a sum of i-x, then by adding x to it, we can get to i.

In simpler terms, to solve for dp[i] which represents the number of combinations to reach a sum of i, we look at all numbers x in

nums that could contribute to i and sum up all the combinations that make up the sum i-x (all of which are valid ways to reach i

In our case, we use an array dp to store the number of combinations that sum up to each value up to target. We initialize dp [0] =

This solution's algorithm can be summarized in the following steps: Define an array dp of length target + 1 and initialize it with zeros. dp[0] is set to 1 since there's one combination that results in a sum of zero, which is not using any numbers.

After the loops finish, dp[target] will hold the number of ways to combine the numbers in nums to sum up to the target. The Python code implementing the solution is as follows:

dp[i] += dp[i - x]

Let's illustrate the solution approach with a smaller example.

Loop through each sub-target i from 1 to target.

For i = 1: The possible number from nums to use is 1.

For i = 2: The possible number from nums to use is 1.

For i = 4: We can use 1, 3, and 4 from nums.

■ If x is less than or equal to i, increment dp[i] by dp[i - x].

Loop through each sub-target i from 1 to target.

Inside this loop, iterate through each number x in nums.

def combinationSum4(self, nums: List[int], target: int) -> int: dp = [1] + [0] * targetfor i in range(1, target + 1): for x in nums:

In this implementation, dp is initialized with size target+1 to accommodate sums from 0 to target inclusive. The two nested loops

are used to populate the dp array according to the dynamic programming strategy outlined above.

if i >= x:

return dp[target]

Example Walkthrough

•

```
Suppose nums = [1, 3, 4] and target = 5. We want to find out how many different combinations of elements from nums can be
added together to sum up to 5.
Here's how we would complete the task step by step:
   Initialize a dp array with target + 1 zeros and set dp[0] to 1, because there is exactly one way to achieve the sum of 0 (using
   no elements).
```

After initialization, our dp array looks like this: dp = [1, 0, 0, 0, 0, 0]

 \circ We update dp[1] to be dp[1] + dp[1 - 1] = dp[1] + dp[0] = 0 + 1 = 1.

For each sub-target i, consider each number x from nums. Let's fill the dp array:

- \circ We update dp[2] to be dp[2] + dp[2 1] = dp[2] + dp[1] = 0 + 1 = 1. For i = 3: We can use 1 and 3 from nums. \circ We can update dp[3] to be dp[3] + dp[3 - 1] (using 1) = dp[3] + dp[2] = 0 + 1 = 1.
- \circ Update dp[4] with 4: dp[4] + dp[4 4] = dp[4] + dp[0] = 2 + 1 = 3. For i = 5: We can use 1, 3, and 4.

 \circ Update dp[5] with 1: dp[5] + dp[5 - 1] = dp[5] + dp[4] = 0 + 3 = 3.

 \circ Update dp[5] with 3: dp[5] + dp[5 - 3] = dp[5] + dp[2] = 3 + 1 = 4.

def combinationSum4(self, nums: List[int], target: int) -> int:

combinations = [1] + [0] * target

for i in range(1, target + 1):

for num in nums:

if i >= num:

print(solution.combinationSum4([1,2,3], 4)) # Output: 7

public int combinationSum4(int[] nums, int target) {

int[] dp = new int[target + 1];

○ Update dp [4] with 3: No change, since dp [4 - 3] is zero.

5 using elements from nums [1, 3, 4] with repetition allowed.

 \circ Update dp[4] with 1 to be dp[4] + dp[4 - 1] = dp[4] + dp[3] = 0 + 2 = 2.

 \circ Update dp[5] with 4: dp[5] + dp[5 - 4] = dp[5] + dp[1] = 4 + 1 = 5. The final dp array is [1, 1, 1, 2, 3, 5], and dp [5] is 5, meaning there are five different combinations to reach the target sum of

• We can also update dp[3] using 3 to be dp[3] + dp[3 - 3] = dp[3] + dp[0] = 1 + 1 = 2.

 \bullet 1 + 1 + 1 + 1 + 1 \bullet 1 + 1 + 3

No other combinations are possible without exceeding the target, so the final answer is 5.

Initialize a list to hold the count of combinations for each value up to the target.

Iterate through each value from 1 to target to find the combinations.

combinations[0] is 1 because there's one way to have a total of 0: by choosing nothing.

For each number in the nums list, check whether it can be used in a combination.

// dp represents the number of combinations to make up each value from 0 up to target

// There is exactly one combination to make up the target 0, which is to choose nothing

// Iterate over all values from 1 to target to find combinations

// Compute the number of combinations for each sum from 1 to target

// and adding would not cause integer overflow

if $(i \ge num \&\& dp[i - num] < INT_MAX - dp[i]) {$

// Return the total number of combinations to reach the target sum

// if the current sum minus the current number is non-negative,

// the combination count of the (current sum - current number)

// Increment the current sum's combination count by

// Check each number in the array to see if it can be used to reach the current sum (i)

for (int i = 1; i <= target; ++i) {</pre>

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

for (int num : nums) {

return dp[target];

If the current number can be used in a combination for the current total (i),

add the number of combinations without this number (i.e., combinations[i - num]).

Python

combinations[i] += combinations[i - num] # Return the number of combinations that add up to the target value. return combinations[target]

solution = Solution()

dp[0] = 1;

Example usage:

class Solution {

Java

Those combinations are:

Solution Implementation

from typing import List

class Solution:

 \bullet 1 + 3 + 1

 \bullet 3 + 1 + 1

• 1 + 4

```
for (int i = 1; i <= target; ++i) {</pre>
           // Iterate through all numbers in the given array
            for (int num : nums) 
                // If the number is less than or equal to the current target (i)
                // then we can use it to form a combination
                if (i >= num) {
                    // Add the number of combinations from the previous value (i - num)
                    // to the current number of combinations
                    dp[i] += dp[i - num];
        // Return the number of combinations to form the target
        return dp[target];
C++
#include <vector>
#include <climits>
class Solution {
public:
    int combinationSum4(vector<int>& nums, int target) {
        // Create a dp vector to store the number of combinations for each value up to the target
        // dp[i] will represent the number of ways to reach the sum i
        vector<int> dp(target + 1, 0);
       // There is one combination to reach the sum of 0 which is by choosing no element
       dp[0] = 1;
```

};

```
TypeScript
  function combinationSum4(nums: number[], target: number): number {
      // Initialize an array of length target + 1 to store the number of ways
      // to reach every number up to target using the given numbers in nums
      const combinationCounts: number[] = new Array(target + 1).fill(0);
      // There is one way to reach 0, which is by using no numbers
      combinationCounts[0] = 1;
      // Loop through all numbers from 1 to target
      for (let currentTarget = 1; currentTarget <= target; ++currentTarget) {</pre>
          // Check each number in nums to see if it can be used to reach currentTarget
          for (const num of nums) {
              // If currentTarget is at least as large as num, it means that
              // num can contribute to a combination that adds up to currentTarget
              if (currentTarget >= num) {
                  // Increase the number of combinations for currentTarget by the number of combinations
                  // that result in the value (currentTarget - num), since num can nicely top up this value
                  combinationCounts[currentTarget] += combinationCounts[currentTarget - num];
      // Return the total number of ways to reach the target using numbers from nums
      return combinationCounts[target];
from typing import List
class Solution:
   def combinationSum4(self, nums: List[int], target: int) -> int:
       # Initialize a list to hold the count of combinations for each value up to the target.
       # combinations[0] is 1 because there's one way to have a total of 0: by choosing nothing.
        combinations = [1] + [0] * target
       # Iterate through each value from 1 to target to find the combinations.
        for i in range(1, target + 1):
            # For each number in the nums list, check whether it can be used in a combination.
            for num in nums:
                # If the current number can be used in a combination for the current total (i),
                # add the number of combinations without this number (i.e., combinations[i - num]).
                if i >= num:
                    combinations[i] += combinations[i - num]
       # Return the number of combinations that add up to the target value.
        return combinations[target]
# Example usage:
```

for i in range(1, target + 1): This loop runs once for every integer from 1 to target, inclusive, so it runs target times.

Inside the outer loop, we have an inner loop:

We have an outer loop:

Time and Space Complexity

solution = Solution()

Time Complexity

print(solution.combinationSum4([1,2,3], 4)) # Output: 7

for x in nums: if i >= x: f[i] += f[i - x]

The time complexity of the given code can be analyzed based on the nested loops.

```
the outer loop.
However, not all iterations of the inner loop will execute the update f[i] += f[i-x]. The condition i >= x needs to be met. In
```

Space Complexity

```
the worst case, though, where i is always greater than or equal to every element x in nums, the body of the inner loop will
execute.
```

```
Analyzing space complexity involves looking at how much additional memory the algorithm uses as a function of the input size.
The space complexity is driven by the list f that has target + 1 elements:
```

f = [1] + [0] * target

This means we need a space proportional to (and linear with) target. Hence, the space complexity of the algorithm is 0(target). No other data structures that scale with the input size is used, so the fixed-size variables and inputs do not affect the overall space complexity significantly compared to the list f which dominates the space usage.

This loop iterates over all elements in nums. If n is the number of elements in nums, the inner loop runs n times for each iteration of

Therefore, the worst-case time complexity is the product of the number of iterations of both loops, which is 0(target * n).