377. Combination Sum IV (Coin change's twin)

Given a list of numbers and a target amount, list the number of permutations of numbers from the list that sum to the target amount.

ex. [1, 2, 3] target = 4

1.) (1,1,1,1)
2.) (1,1,2) 5.) (2,1,1) 6.) (1,2,1)
3.) (1,3) 7.) (3,1)
4.) (2,2)

Brute Force Approach:

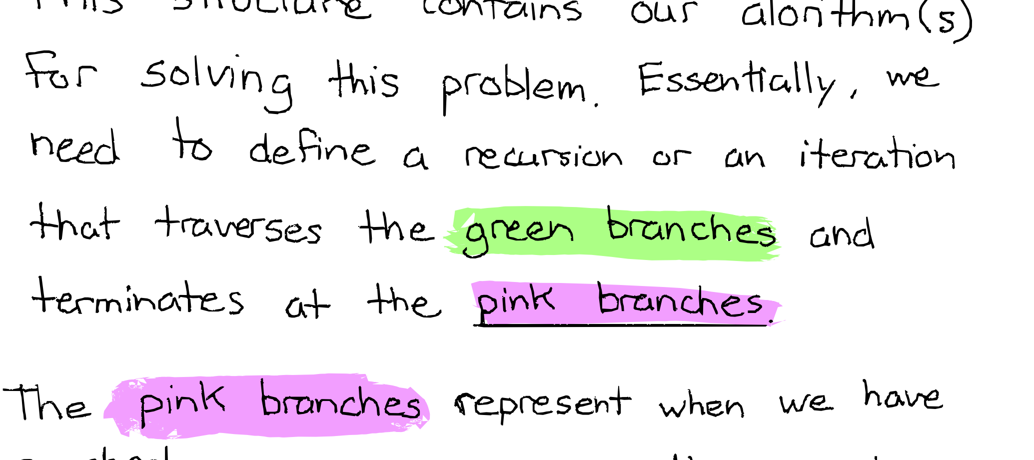
When we sum values, we know we have reached our target when our running difference equals the last value we are looking at.

e.g. $4 - (2+1) - 1 = 0$ ← means we have reach our target

target — values we have already looked at — potential last value

This is a bit odd way to look at it but it will help with understanding the ultimate optimal solution.

To brute force we have to look at each combination until we have a sum that is ≥ target. It ends up looking like a branching tree:



卌 II = 7 permutations

This structure contains our alorithm(s) for solving this problem. Essentially, we need to define a recursion or an iteration that traverses the green branches and terminates at the pink branches.

The pink branches represent when we have reached our invalid basecase. Here, our base case is when our sum is greater than the target amount.

The green branches represent when we have reache our base case where our sum equals the target value. When this happens we know we've found a valid sum and need to add 1 to our total.

So how does this look with recursion?:

```
def sumIV(target, nums):
    def helper( t ):
        rslt = 0
        if t < 0: return 0
        if t == 0: return 1
        for num in nums:
            if num ≤ t:
                rslt += helper( t - num)
    return helper(target)
```

we can avoid a stack frame allocation

```
def combinationSumIV(self, nums:list[int], target:int) -> int :
    def helper(t:int) -> int:
        if t == 0:
            return 1
        rslt = 0
        for num in nums:
            if num <= t:
                rslt += helper(t-num)
        return rslt
    return helper(target)
```

This is cool, but we do a lot of the same calculations over and over again. Every time we recurse on a value t we have already seen, we are recomputing a result that was already solved.

This is where memoization comes in. Before we recurse and go back down the rabbit hole again, lets check to see if we haven't already solved this problem before.

We do this by, likewise, hashing our subproblem results:

```
def combinationSumIV(self, nums:list[int], target:int) -> int :
    memo:dict[int,int] = {}
    def helper(t:int) -> int:
        if t == 0:
            return 1
        rslt = 0
        for num in nums:
            if (t-num) in memo: # check to see if we have it memoized already
                rslt += memo[t-num]
            elif num <= t:
                rslt += helper(t-num)
        memo[t] = rslt # memo our rslt
        return rslt

    return helper(target)
```

Sweet! But now we run into a different issue... Memory. We have significantly cut down on compute time, but we are still at risk of running out of memory through recursion. If only there was a way to not have to check every single combination. Completely avoiding the need to recurse at all even. There is!

To solve this recursively is to solve from the top going down until we have discovered the solution of the smallest subproblem (base cases). If we already know what to expect at our base case, could we instead solve the bottom first and work our way up?

We can achieve this via dynamic programming.

First we define a dp table from which we will track our sub problem solution result.

In this case, we are trying to reach a target value. So let's create an array of that will track our results from 0 up to the target value. We init it to 0 first

target = 4 => dp = [0,0,0,0,0]

This intuition comes from the idea that in order to find the number of permutations for target x we need to find the number of permutations for target x - 1. We can extrapolate this until our target is 0. At which (from a list of positive integess), there is only 1 way to get our target value: the empty set of numbers (this is by definition from combinatorics). So we can set the 0th index of of dp array to 1:

dp = [1, 0, 0, 0, 0]

From here we will iterate from 1 to our target building up our dp table until we reach our target value:

```
for x in range(1, target + 1):
    for n in nums:
        if n ≤ x:
            dp[x] += dp[x-n]

return dp[target]
```

Suddenly we are in linear time! O(n)