L377/P16.4.Combination Sum IV(Backpack VI) DP Solutions (Refer L39.Combination Sum)

<https://leetcode.com/problems/combination-sum-iv/>

**Solution 4: Bottom Up 2D DP and 1D DP [Standard 0-1 Knapsack problem] (60 min)**

**2D DP**

class Solution {

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

// dp[i][j] means the combination count of use up to j elements to get target i

int[][] dp = new int[target + 1][candidates.length + 1];

// No matter how many nums given only 1 way to make target=0 (no way)

for(int i = 0; i <= candidates.length; i++) {

dp[0][i] = 1;

}

// As defined only positive elements (1 <= nums[i] <= 1000), for target = 0

// dp always keep as 0, no explicit declare required

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

// dp[i][0] = 0;

//}

// Outside for loop as 'target', inside for loop as ''numbers', to find permutation

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

for(int j = 1; j <= candidates.length; j++) {

// If we don't pick up current number, then current combination count depends

// on previous (j - 1) elements only, no new combination introduced

dp[i][j] = dp[i][j - 1];

// If we pick up current number, new combination introduced, new count is

// dp[i - candidates[j - 1]][candidates.length]

if(i >= candidates[j - 1]) {

dp[i][j] += dp[i - candidates[j - 1]][candidates.length];

}

}

}

return dp[target][candidates.length];

}

}

**Refer to**

<https://leetcode.com/problems/combination-sum-iv/solutions/702432/java-1d-2d-bottom-up-top-down/>

As bottom up are meant to be start with the base cases and the fill the transition table.

Here the base case is if the amount is 0, no matter how many nums given only 1 way to make 0 (no way)

In 2D bottom up last col of each row contains the number of ways to make the 'row' amount.

Understand this transition for top down to bottom up.

--------------------------------------------------------------------------------------------

In bottom up dp there is no rule involved. Its just that the values computed before can be used. In this problem for each row(amount)

different coins are being used so the last col signifies the given amount using all the coins. Now for a new amount we need those ways.

Example for amount = 5 and 1, 2, 3

1

11 | 2

111, 21 | 12 | 3

1111, 211, 121, 31 | 112, 22 | 13

11111, 2111, 1211, 311, 1121, 221, 131 | 1112, 212, 122, 32 | 113, 23

In the above example if you notice. row no. 4 has 7 ways

Now to create the 5th row what we do is concatenate the coins and see if it is less than the curr target

1111, 211, 121, 31 | 112, 22 | 13 ------>(using coin 1)------>11111, 2111, 1211, 311, 1121, 221, 131

111, 21 | 12 | 3------>(using coin 2)------>1112, 212, 122, 32

11 | 2------>(using coin 3)------>113, 23

--------------------------------------------------------------------------------------------

In this problem we are typically looking for "Permutations" because the order of numbers inside any given arrangement matters, so as an example, given these 3 arrangements => [123, 132, 231] we are using same numbers but with diff. order!! and we still count them distinctly as 3 diff. ways! and this is the main difference(but not the only one) between a combination and a permutation, combinations on the other side would have considered all our 3 arrangements as only 1 way of arranging numbers because in combinations order doesn't matter while in permutations it does matter.

Coming back to this problem after this brief intro. if we want to re-use a permutation of a given sum that we already calculated before, it will be located in the last cell of that sum where we have used all our given items(remember a permutation is only valid if it consists of all numbers in the given input array)

i.e.

**dp[i][j] means the combination count of use up to j elements to get target i**

dp[i - coins[j - 1]][coins.length], where: i - coins[j - 1]: will get us to the row of the previous sum we are looking for [coins.length]: last column where we exhausted all our given items and this cell have the permutations count for this sum.

I also want to clear something out, when using 2D space to solve this problem, it doesn't matter if you used rows to represent sum and columns for items or vice versa, both will work just fine if you make sure for each sum you consume all numbers before moving on to the next sum

--------------------------------------------------------------------------------------------

**In bottom up 2D-DP solution how to understand dp[i][j] = dp[i][j] + dp[i - nums[j - 1]][nums.length] ?**

Why dp[i][j] = dp[i][j] + dp[i - nums[j - 1]][nums.length] ? usually it suppose to be dp[i][j] = dp[i][j] + dp[i - nums[j - 1]][j]

e.g

dp[i][j] means the combination count of use up to j element to get target i

i = 4, j = 1 -> which means target = 4, use up to first one elements of nums

(1)if not use the first element in nums(first element is nums[0] = 1, not use 1)

dp[4][1] = dp[4][1 - 1] = 0

(2)if use the first element in nums(first element is nums[0] = 1, use 1)

if(4 >= nums[1 - 1]) -> true

use correct formula:

dp[i][j] = dp[i][j] + dp[i - nums[j - 1]][nums.length]

dp[4][1] = d[4][1] + dp[4 - nums[1 - 1]][3] = dp[4][1] + dp[3][3] = 4

use wrong formula what will happen ?

dp[i][j] = dp[i][j] + dp[i - nums[j - 1]][j]

dp[4][1] = d[4][1] + dp[4 - nums[1 - 1]][1] = dp[4][1] + dp[3][1] = 2

What's the difference ?

For correct answer

dp[4][1] = d[4][1] + dp[4 - nums[1 - 1]][3] = dp[4][1] + dp[3][3] = 4

because we try to get target = 4 by only using first element as 1(dp[4][1]) but based on

previously calculated all permutations for target = 3(dp[3][3]) "by using all elements",

the critical part is "by using all elements", under this case, only last column

value for target = 3 means "using all elements"

1

11 | 2

111, 21 | 12 | 3 -> target = 3 "using all elements" have 4 ways => this is the value on last column at row = 4 (dp[4][3])

now by using only first element as 1 to get target 4, it will be also 4 ways as below

3rd row all 4 ways plus first element as 1

111 + 1, 21 + 1, 12 + 1, 3 + 1

=> 1111, 211, 121, 31

Note:

If we extend as "using only first two elements as 1 and 2 to get target 4", it will be 6 ways as below

3rd row all 4 ways plus first element as 1 have 4 ways (detail above)

2nd row all 2 ways plus second element as 2 have 2 ways (detail above)

11 + 2, 2 + 2

=> 112, 22

If we extend as "using all three elements as 1, 2 and 3 to get target 4", it will be 7 ways as below

3rd row all 4 ways plus first element as 1 have 4 ways (detail above)

2nd row all 2 ways plus second element as 2 have 2 ways (detail above)

1st row all 1 way plus third element as 3 have 1 way (detail above)

1 + 3

=> 13

=================================================================

How about the wrong answer dp[i - nums[j - 1]][j] = dp[3][1] = 2 ?

that means by using only first element as 1 but based on previously calculated permutations

for target = 3 "by using only first element as 1", its not get target = 3 "by using all elements",

which means currently we have target = 2 "by using all elements" as 2 ways (dp[3][3]), based on

it to get target = 3 "by using only first element as 1", will be only 2 ways

11 | 2 -> target = 2 "using all elements" have 2 ways => this is the value on last column at row = 3 (dp[3][3])

now by using only first element as 1 to get target 3, it will be also 2 ways as below

11 + 1, 2 + 1

=> 111, 21

then based on these 2 ways to get target = 3, still "by using only first element as 1" to get target = 4

111 + 1, 21 + 1

=> 1111, 211

that's how 2 ways for target = 4 comes from

**1D DP**

class Solution {

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

// dp[i] means how many ways to get 'target' with all elements

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

// For target = 0, just take no element is the only solution

dp[0] = 1;

// Outside for loop as 'target', inside for loop as ''numbers', to find permutation

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

for(int j = 0; j < nums.length; j++) {

// When element(nums[j]) is feasible(<= target 'i'), we can infer an

// increment count as dp[i - nums[j]]

if(i >= nums[j]) {

dp[i] += dp[i - nums[j]];

}

}

}

return dp[target];

}

}

**What's the difference between L518.Coin Change II and L.377 Combination Sum IV on Bottom Up DP solution ?**

**Permutation v.s. Combination for loop matters ?**

**Refer to**

<https://leetcode.com/problems/combination-sum-iv/discuss/85036/1ms-Java-DP-Solution-with-Detailed-Explanation/191809>

Some comment about the iterative solution with different orders of loops:

**Order-1: combinations considering different sequences (Permutations)**

for each sum in dp[]

for each num in nums[]

if (sum >= num)

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

**Order-2: combinations NOT considering different sequences**

for each num in nums[]

for each sum in dp[] >= num

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

Order-1 is used to calculate the number of combinations considering different sequences

Order-2 is used to calculate the number of combinations NOT considering different sequences

Give an example nums[] = {1, 2, 3}, target = 4

Order-1 considers the number of combinations starting from 1, 2, and 3, respectively, so all sequences are considered as the graph below.

1 --> 1 --> 1 --> 1 --> (0)

1 --> 1 --> 2 --> (0)

1 --> 2 --> 1 --> (0)

1 --> 3 --> (0)

2 --> 1 --> 1 --> (0)

2 --> 2 --> (0)

3 --> 1 --> (0)

Order-2 considers the number of combinations starting from 0 (i.e., not picking anyone), **and the index of the num picked next must be >= the index of previous picked num, so different sequences are not considered**, as the graph below.

(0) --> 1 --> 1 --> 1 --> 1

(0) --> 1 --> 1 --> 2

(0) --> 1 --> 3

(0) --> 2 --> 2

**Follow Up: What if negative numbers are allowed in the given array? How does it change the problem? What limitation we need to add to the question to allow negative numbers?**

class Solution {

// The new parameter 'maxLen' must introduced to control length

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

if(nums == null || nums.length == 0 || target <= 0 || maxLen <= 0) {

return 0;

}

// key = current target, value = <key = current length, value = combinations mapping to current length and current target>

Map<Integer, Map<Integer, Integer>> map = new HashMap<Integer, Map<Integer, Integer>>();

return helper(nums, target, map, 0, maxLen);

}

private int helper(int[] nums, int sum, Map<Integer, Map<Integer, Integer>> map, int len, int maxLen) {

if(len > maxLen) {

return 0;

}

if(sum == 0) {

return 1;

}

if(map.containsKey(sum) && map.get(sum).containsKey(len)) {

return map.get(sum).get(len);

}

int result = 0;

for(int i = 0; i < nums.length; i++) {

if(sum >= nums[i]) {

result += helper(nums, sum - nums[i], map, len + 1, maxLen);

}

}

if(!map.containsKey(sum)) {

map.put(sum, new HashMap<Integer, Integer>());

}

Map<Integer, Integer> memo = map.get(sum);

memo.put(len, result);

return result;

}

}

**Refer to**

<https://leetcode.com/problems/combination-sum-iv/solutions/85038/java-follow-up-using-recursion-and-memorization/>

**In order to allow negative integers, the length of the combination sum needs to be restricted, or the search will not stop. This is a modification from my [previous solution](https://discuss.leetcode.com/topic/52255/java-recursion-solution-using-hashmap-as-memory), which also use memory to avoid repeated calculations.**

Map<Integer, Map<Integer,Integer>> map2 = new HashMap<>();

private int helper2(int[] nums, int len, int target, int MaxLen) {

int count = 0;

if ( len > MaxLen ) return 0;

if ( map2.containsKey(target) && map2.get(target).containsKey(len)) {

return map2.get(target).get(len);

}

// Add condition "&& len >= 1" explicitly means negative element exist

if ( target == 0 && len >= 1) count++;

for (int num: nums) {

count+= helper2(nums, len+1, target-num, MaxLen);

}

if ( ! map2.containsKey(target) ) map2.put(target, new HashMap<Integer,Integer>());

Map<Integer,Integer> mem = map2.get(target);

mem.put(len, count);

return count;

}

public int combinationSum42(int[] nums, int target, int MaxLen) {

if (nums == null || nums.length ==0 || MaxLen <= 0 ) return 0;

map2 = new HashMap<>();

return helper2(nums, 0,target, MaxLen);

}