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

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

Given an array of **distinct** integers nums and a target integer target, return *the number of possible combinations that add up to* target.

The test cases are generated so that the answer can fit in a **32-bit** integer.

**Example 1:**

Input: nums = [1,2,3], target = 4

Output: 7

Explanation:

The possible combination ways are:

(1, 1, 1, 1)

(1, 1, 2)

(1, 2, 1)

(1, 3)

(2, 1, 1)

(2, 2)

(3, 1)

Note that different sequences are counted as different combinations.

**Example 2:**

Input: nums = [9], target = 3

Output: 0

**Constraints:**

* 1 <= nums.length <= 200
* 1 <= nums[i] <= 1000
* All the elements of nums are **unique**.
* 1 <= target <= 1000

**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?

**Attempt 1: 2023-01-13**

**Solution 1: Backtracking (1200 min)**

**Important note: Backtracking not only work for object type, but also primitive type (e.g the integer parameter 'target' for recursion function also applicable for backtracking)**

**Wrong solution: Why 'target += candidates[i]' is required ?**

class Solution {

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

if(candidates == null || candidates.length == 0) {

return 0;

}

// Base case

if(target == 0) {

return 1;

}

int res = 0;

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

if(target - candidates[i] >= 0) {

// This is a wrong way, test out by {1,2,3},4 -> output:3, expect:7.

// Because we suppose NOT change current level 'target' during the recursion,

// we ONLY want to send the NEW 'target' as "target - candidates[i]", to next

// recursion level.

// Let's say if we must put "target -= candidates" before "res += test2(candidates, target)"

// then based on classical backtracking style, since current level variable 'target' changed

// by "target -= candidates[i]", we have to explicitly make it up by "target += candidates"

target -= candidates[i];

res += combinationSum4(candidates, target);

// target += candidates[i];

}

}

return res;

}

}

**Correct Backtracking Style 1 (TLE):**

**Backtracking also applicable for primitive type 'parameter' that passed into recursion method, fix the wrong solution issue by adding 'backtracking' statement "target += candidates[i]"**

class Solution {

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

if(candidates == null || candidates.length == 0) {

return 0;

}

// Base case

if(target == 0) {

return 1;

}

int res = 0;

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

if(target - candidates[i] >= 0) {

target -= candidates[i];

res += combinationSum4(candidates, target);

target += candidates[i];

}

}

return res;

}

}

Time Complexity : O(2^N)

Space Complexity: O(2^N)

**Correct Backtracking Style 2 (TLE):**

**If we introduce additional parameter 'index' for recursion method**

class Solution {

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

if(candidates == null || candidates.length == 0) {

return 0;

}

return helper(candidates, target, 0);

}

private int helper(int[] candidates, int target, int index) {

// 'index' not actually used in later for loop which including recursion,

// in for loop it use local variable 'i' to always iterate from index = 0

// in any recursion level, which enable the ability to go back to previous

// elements in any recursion level, because element order matters in L377,

// not like L39 order doesn't matter which requires local variable 'i' start

// with previous recursion level passed in value 'index' for no move back

if(index >= candidates.length) {

return 0;

}

// Our goal: when currentSum = target

if(target == 0) {

return 1;

}

int res = 0;

// Our choices: We can choose a number from the list any number of times and all the numbers

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

// Our constraints: We can't go beyond target, we can take more element than available in array

if(target - candidates[i] >= 0) {

target -= candidates[i];

res += helper(candidates, target, i);

target += candidates[i];

}

}

return res;

}

}

Time Complexity : O(2^N)

Space Complexity: O(2^N)

**We can monitor the backtracking steps by printing logs**

public class Solution {

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

if(candidates == null || candidates.length == 0) {

return 0;

}

return backtrack\_logging(candidates, 0, target, 0);

//return helper(candidates, 0, target);

//List<List<Integer>> result = new ArrayList<List<Integer>>();

//int tmp = helper2(candidates, 0, target, new ArrayList<Integer>(), result);

//System.out.println(result);

//return tmp;

}

private int backtrack\_logging(int[] candidates, int index, int target, int recursionLevel) {

System.out.println("Enter -----------------------------------------------------");

System.out.println("Recursion level: " + recursionLevel + ", target: " + target);

if(index >= candidates.length) {

System.out.println("Exit with 0");

return 0;

}

// Our goal: when currentSum = target

if(target == 0) {

System.out.println("Exit with 1");

return 1;

}

int res = 0;

// Our choices: We can choose a number from the list any number of times and all the numbers

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

// Our constraints: We can't go beyond target, we can take more element than available in array

if(target - candidates[i] >= 0) {

target -= candidates[i];

System.out.println("Backtrack start => Recursion level: " + recursionLevel + ", target decrease from: " + (target + candidates[i]) + " to " + target);

res += backtrack\_logging(candidates, i, target, recursionLevel + 1);

//backtrack

target += candidates[i];

System.out.println("Backtrack done => Recursion level: " + recursionLevel + ", target increase from: " + (target - candidates[i]) + " to " + target);

}

}

System.out.println("Exit with res " + res);

return res;

}

public static void main(String[] args) {

Solution s = new Solution();

int result = s.combinationSum4(new int[]{1,2,3}, 4);

System.out.println(result);

}

}

**Logs**

Enter -----------------------------------------------------

Recursion level: 0, target: 4

Backtrack start => Recursion level: 0, target decrease from: 4 to 3

Enter -----------------------------------------------------

Recursion level: 1, target: 3

Backtrack start => Recursion level: 1, target decrease from: 3 to 2

Enter -----------------------------------------------------

Recursion level: 2, target: 2

Backtrack start => Recursion level: 2, target decrease from: 2 to 1

Enter -----------------------------------------------------

Recursion level: 3, target: 1

Backtrack start => Recursion level: 3, target decrease from: 1 to 0

Enter -----------------------------------------------------

Recursion level: 4, target: 0

Exit with 1

Backtrack done => Recursion level: 3, target increase from: 0 to 1

Exit with res 1

Backtrack done => Recursion level: 2, target increase from: 1 to 2

Backtrack start => Recursion level: 2, target decrease from: 2 to 0

Enter -----------------------------------------------------

Recursion level: 3, target: 0

Exit with 1

Backtrack done => Recursion level: 2, target increase from: 0 to 2

Exit with res 2

Backtrack done => Recursion level: 1, target increase from: 2 to 3

Backtrack start => Recursion level: 1, target decrease from: 3 to 1

Enter -----------------------------------------------------

Recursion level: 2, target: 1

Backtrack start => Recursion level: 2, target decrease from: 1 to 0

Enter -----------------------------------------------------

Recursion level: 3, target: 0

Exit with 1

Backtrack done => Recursion level: 2, target increase from: 0 to 1

Exit with res 1

Backtrack done => Recursion level: 1, target increase from: 1 to 3

Backtrack start => Recursion level: 1, target decrease from: 3 to 0

Enter -----------------------------------------------------

Recursion level: 2, target: 0

Exit with 1

Backtrack done => Recursion level: 1, target increase from: 0 to 3

Exit with res 4

Backtrack done => Recursion level: 0, target increase from: 3 to 4

Backtrack start => Recursion level: 0, target decrease from: 4 to 2

Enter -----------------------------------------------------

Recursion level: 1, target: 2

Backtrack start => Recursion level: 1, target decrease from: 2 to 1

Enter -----------------------------------------------------

Recursion level: 2, target: 1

Backtrack start => Recursion level: 2, target decrease from: 1 to 0

Enter -----------------------------------------------------

Recursion level: 3, target: 0

Exit with 1

Backtrack done => Recursion level: 2, target increase from: 0 to 1

Exit with res 1

Backtrack done => Recursion level: 1, target increase from: 1 to 2

Backtrack start => Recursion level: 1, target decrease from: 2 to 0

Enter -----------------------------------------------------

Recursion level: 2, target: 0

Exit with 1

Backtrack done => Recursion level: 1, target increase from: 0 to 2

Exit with res 2

Backtrack done => Recursion level: 0, target increase from: 2 to 4

Backtrack start => Recursion level: 0, target decrease from: 4 to 1

Enter -----------------------------------------------------

Recursion level: 1, target: 1

Backtrack start => Recursion level: 1, target decrease from: 1 to 0

Enter -----------------------------------------------------

Recursion level: 2, target: 0

Exit with 1

Backtrack done => Recursion level: 1, target increase from: 0 to 1

Exit with res 1

Backtrack done => Recursion level: 0, target increase from: 1 to 4

Exit with res 7

7

Process finished with exit code 0

**Why for(int i = index; i < ...) is wrong in L377.Combination Sum IV(in L39.Combination Sum it was)?**

**In another word, Permutation (L377) to start for loop with i = 0, Combination (L39) to start for loop with i = index**

L377.Combination Sum IV

Test: {1,2,3},4, output=4, expect=7

[[1, 1, 1, 1], [1, 1, 2], [1, 3], [2, 2]] -> Wrong for L377.Combination Sum IV, correct for L39.Combination Sum

4

[[1, 1, 1, 1], [1, 1, 2], [1, 2, 1], [1, 3], [2, 1, 1], [2, 2], [3, 1]]

7

**If 'i' start with index, 'i' will never go back to previous index, e.g if current level recursion index is 1, it never go back to index 0 for next level recursion**

**In L39.Combination Sum requires return a list of all unique combinations of candidates even same number may be chosen from candidates an unlimited number of times**

**In L377.Combination Sum IV the difference is order of combination matters, in L39 for {1,2,3}, target = 4, combination (1,1,2) treat as same combination as (1,2,1), but in L377, these two combinations are different even both have two '1' and one '2' since order changed**

**We can monitor the combination changes by adding list to observe both L39 and L377 combination changes**

L39. Combination Sum -> for(int i = index; i < ...)

-> [[1, 1, 1, 1], [1, 1, 2], [1, 3], [2, 2]]

{ }

/ | \

{1} {2} {3}

/ \ | \

{1,1} {1,3} {2,2} x

/ \

{1,1,1} {1,1,2}

/

{1,1,1,1}

class Solution {

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

if(candidates == null || candidates.length == 0) {

return 0;

}

List<List<Integer>> result = new ArrayList<List<Integer>>();

int tmp = L39\_i\_start\_with\_index(candidates, 0, target, new ArrayList<Integer>(), result);

System.out.println(result);

return tmp;

}

private int L39\_i\_start\_with\_index(int[] candidates, int index, int target, List<Integer> list, List<List<Integer>> result) {

if (index >= candidates.length)

return 0;

// Our goal: when currentSum = target

if (0 == target) {

result.add(new ArrayList<Integer>(list));

return 1;

}

int res = 0;

// Our choices: We can choose a number from the list any number of times and all the numbers

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

// Our constraints: We can't go beyond target, we can take more element than available in array

if (target - candidates[i] >= 0) {

target -= candidates[i];

list.add(candidates[i]);

res += L39\_i\_start\_with\_index(candidates, i, target, list, result);

target += candidates[i];

list.remove(list.size() - 1);

}

}

return res;

}

}

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

L377. Combination Sum IV -> for(int i = 0; i < ...)

-> [[1, 1, 1, 1], [1, 1, 2], [1, 2, 1], [1, 3], [2, 1, 1], [2, 2], [3, 1]]

{ }

/ | \

{1} {2} {3}

/ | \ / \ \

{1,1} {1,2} {1,3} {2,1} {2,2} {3,1}

/ \ | |

{1,1,1} {1,1,2}{1,2,1} {2,1,1}

/

{1,1,1,1}

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

class Solution {

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

if(candidates == null || candidates.length == 0) {

return 0;

}

List<List<Integer>> result = new ArrayList<List<Integer>>();

int tmp = L377\_i\_start\_with\_0(candidates, 0, target, new ArrayList<Integer>(), result);

System.out.println(result);

return tmp;

}

private int L377\_i\_start\_with\_0(int[] candidates, int index, int target, List<Integer> list, List<List<Integer>> result) {

if (index >= candidates.length)

return 0;

// Our goal: when currentSum = target

if (0 == target) {

result.add(new ArrayList<Integer>(list));

return 1;

}

int res = 0;

// Our choices: We can choose a number from the list any number of times and all the numbers

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

// Our constraints: We can't go beyond target, we can take more element than available in array

if (target - candidates[i] >= 0) {

target -= candidates[i];

list.add(candidates[i]);

res += L377\_i\_start\_with\_0(candidates, i, target, list, result);

target += candidates[i];

list.remove(list.size() - 1);

}

}

return res;

}

}

**And for L377.Combination Sum IV since we don't have usage of 'index' as for(int i = index; i <...), there is no need to add 'index' as part of parameter**

class Solution {

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

if(candidates == null || candidates.length == 0) {

return 0;

}

List<List<Integer>> result = new ArrayList<List<Integer>>();

int tmp = L377\_i\_start\_with\_0\_no\_index\_required(candidates, target, new ArrayList<Integer>(), result);

System.out.println(result);

return tmp;

}

private int L377\_i\_start\_with\_0\_no\_index\_required(int[] candidates, int target, List<Integer> list, List<List<Integer>> result) {

// Our goal: when currentSum = target

if (target == 0) {

result.add(new ArrayList<Integer>(list));

return 1;

}

int res = 0;

// Our choices: We can choose a number from the list any number of times and all the numbers

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

// Our constraints: We can't go beyond target, we can take more element than available in array

if (target - candidates[i] >= 0) {

target -= candidates[i];

list.add(candidates[i]);

res += L377\_i\_start\_with\_0\_no\_index\_required(candidates, target, list, result);

target += candidates[i];

list.remove(list.size() - 1);

}

}

return res;

}

}

**Bottle neck: 2^n recursion trees. If you draw this tree, you'll see many overlapping sub-problems, to resolve TLE, use Top Down DP to cache the recursion**

**Top Down DP Backtracking + Memo Style 1:**

**Still with non-useful 'index' version, build memo based on each 'target' (not based on non-useful 'index')**

class Solution {

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

if(candidates == null || candidates.length == 0) {

return 0;

}

// Build memo based on each 'target' (not based on non-useful 'index')

Integer[] memo = new Integer[target + 1];

return helper(candidates, target, 0, memo);

}

private int helper(int[] candidates, int target, int index, Integer[] memo) {

// 'index' not actually used in later for loop which including recursion,

// in for loop it use local variable 'i' to always iterate from index = 0

// in any recursion level, which enable the ability to go back to previous

// elements in any recursion level, because element order matters in L377,

// not like L39 order doesn't matter which requires local variable 'i' start

// with previous recursion level passed in value 'index' for no move back

if(index >= candidates.length) {

return 0;

}

// Our goal: when currentSum = target

if(target == 0) {

return 1;

}

if(memo[target] != null) {

return memo[target];

}

int res = 0;

// Our choices: We can choose a number from the list any number of times and all the numbers

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

// Our constraints: We can't go beyond target, we can take more element than available in array

if(target - candidates[i] >= 0) {

target -= candidates[i];

res += helper(candidates, target, i, memo);

target += candidates[i];

}

}

memo[target] = res;

return res;

}

}

Time Complexity : O(N) ~ O(2^N)

Space Complexity: O(N) ~ O(2^N)

**Top Down DP Backtracking + Memo Style 2:**

**No non-useful 'index' version, build memo based on each 'target' (not based on non-useful 'index')**

class Solution {

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

if(candidates == null || candidates.length == 0) {

return 0;

}

Integer[] memo = new Integer[target + 1];

return helper(candidates, target, memo);

}

private int helper(int[] candidates, int target, Integer[] memo) {

// Our goal: when currentSum = target

if(target == 0) {

return 1;

}

if(memo[target] != null) {

return memo[target];

}

int res = 0;

// Our choices: We can choose a number from the list any number of times and all the numbers

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

// Our constraints: We can't go beyond target, we can take more element than available in array

if(target - candidates[i] >= 0) {

target -= candidates[i];

res += helper(candidates, target, memo);

target += candidates[i];

}

}

memo[target] = res;

return res;

}

}

Time Complexity : O(N) ~ O(2^N)

Space Complexity: O(N) ~ O(2^N)

**We can monitor how memo change behavior on native backtracking recursion (impact both Time Complexity and Space Complexity)**

public class Solution {

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

if(candidates == null || candidates.length == 0) {

return 0;

}

Integer[] memo = new Integer[target + 1];

List<List<Integer>> result = new ArrayList<List<Integer>>();

int tmp = how\_memo\_works(candidates, target, new ArrayList<Integer>(), result, memo);

System.out.println(result);

return tmp;

}

private int how\_memo\_works(int[] candidates, int target, List<Integer> list, List<List<Integer>> result, Integer[] memo) {

// Our goal: when currentSum = target

if (target == 0) {

result.add(new ArrayList<Integer>(list));

return 1;

}

if(memo[target] != null) {

return memo[target];

}

int res = 0;

// Our choices: We can choose a number from the list any number of times and all the numbers

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

// Our constraints : We can't go beyond target, we can take more element than available in array

if (target - candidates[i] >= 0) {

target -= candidates[i];

list.add(candidates[i]);

res += how\_memo\_works(candidates, target, list, result, memo);

target += candidates[i];

list.remove(list.size() - 1);

}

}

memo[target] = res;

return res;

}

public static void main(String[] args) {

Solution s = new Solution();

int result = s.combinationSum4(new int[]{1,2,3}, 4);

System.out.println(result);

}

}

**Why [1, 2, 1], [2, 1, 1], [2, 2], [3, 1] are removed from printing list after adding memo ?**

Without memo

[[1, 1, 1, 1], [1, 1, 2], [1, 2, 1], [1, 3], [2, 1, 1], [2, 2], [3, 1]]

7

With memo

[[1, 1, 1, 1], [1, 1, 2], [1, 3]]

7

Difference

[1, 2, 1], [2, 1, 1], [2, 2], [3, 1] -> the combination not recorded in result

**The combination not recorded in result, because in code 'result' only adding new combination when target = 0 as below:**

if (target == 0) {

result.add(new ArrayList<Integer>(list));

return 1;

}

**But if we adding memo check, sometimes when find existing target's memo, we don't have to continue recursion till target = 0, it directly return memoized result, if(target == 0) logic not touched, result not add new combination**

if(memo[target] != null) {

return memo[target];

}

**Step by step graph**

create memo[4]=7<-{ }-> no call memo[4] as already return final result

/ | \

create memo[3]=4<-{1} {2}---->call memo[2]=2 {3}--->call memo[1]=1

/ | \ / \ return directly \ return directly

create memo[2]=2<-{1,1} {1,2}->call memo[1]=1 {1,3} {2,1} {2,2} hence no touch {3,1} hence no touch

/ \ | return directly | on logic adding on logic adding

create memo[1]=1<-{1,1,1} {1,1,2}{1,2,1} hence no touch {2,1,1} {2,1,1} and {2,2} {3,1} to result

/ on logic adding to result

{1,1,1,1} {1,2,1} to result

**Solution 2: Relation based (60 min)**

**Based on Solution 1 backtracking style, we can promote the solution by removing "backtracking" statement and roll out Solution 2 as "Relation based".**

**Technically for primitive type parameter as int 'target' used in recursion method following "passed by value" mechanism, we don't "passed by reference", hence the new 'target' value (target - candidates[i]) scope only limited for next recursion level based on two premises:**

**1. If new 'target' value as "target - candidates[i]" only happening on recursion method parameter placeholder**

**2. If no explicit statement modify current level 'target' value (even it is a primitive type value)**

target -= candidates[i]; // --> wrong statement modify current level primitive paramter 'target' and no explicit backtracking deduce the wrong solution

res += combinationSum4(candidates, target); // we passed new target value as "target - candidates[i]" to next recursion level but since previous statement the target value also changed in parent level

**Removing "backtracking" statement**

class Solution {

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

if(candidates == null || candidates.length == 0) {

return 0;

}

return helper(candidates, target);

}

private int helper(int[] candidates, int target) {

// Our goal: when currentSum = target

if(target == 0) {

return 1;

}

int res = 0;

// Our choices: We can choose a number from the list any number of times and all the numbers

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

// Our constraints: We can't go beyond target, we can take more element than available in array

if(target - candidates[i] >= 0) {

res += helper(candidates, target - candidates[i]);

}

}

return res;

}

}

**Adding 1D memo as Top Down DP**

class Solution {

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

if(candidates == null || candidates.length == 0) {

return 0;

}

Integer[] memo = new Integer[target + 1];

return helper(candidates, target, memo);

}

private int helper(int[] candidates, int target, Integer[] memo) {

// Our goal: when currentSum = target

if(target == 0) {

return 1;

}

if(memo[target] != null) {

return memo[target];

}

int res = 0;

// Our choices: We can choose a number from the list any number of times and all the numbers

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

// Our constraints: We can't go beyond target, we can take more element than available in array

if(target - candidates[i] >= 0) {

res += helper(candidates, target - candidates[i], memo);

}

}

memo[target] = res;

return res;

}

}

**Refer to**

<https://leetcode.com/problems/combination-sum-iv/solutions/372950/tle-to-100-beat-optimisation-step-by-step-7-solutions/>

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

Big problem above is," backtracking" step. How do we should avoid it?

Think about the recurrence relation first. How does the # of combinations of the target related to the # of combinations of numbers that are smaller than the target?

So we know that target is the sum of numbers in the array. Imagine we only need one more number to reach target, this number can be any one in the array, right? So the # of combinations of target, comb[target] = sum(comb[target - nums[i]]), where 0 <= i < nums.length, and target >= nums[i].

In the example given, we can actually find the # of combinations of 4 with the # of combinations of 3(4 - 1), 2(4- 2) and 1(4 - 3). As a result, comb[4] = comb[4-1] + comb[4-2] + comb[4-3] = comb[3] + comb[2] + comb[1].

Then think about the base case. Since if the target is 0, there is only one way to get zero, which is using 0, we can set comb[0] = 1.

EDIT: The problem says that target is a positive integer that makes me feel it's unclear to put it in the above way. Since target == 0 only happens when in the previous call, target = nums[i], we know that this is the only combination in this case, so we return 1.

Hence;

combination[target] = Sum {combination[target-nums[i]] } where target>=nums[i]

in our example:

combination[4] = combination[4-1] + combination[4-2] + combination[4-3] = combination[3] + combination[2] + combination[1].

Now, what is the base case; When target = 0; in this case, if we simply choose no numbers from nums, then answer would be 1;combination[0] = 1

Our relationship is

Recurrence relation

combination[target] = {

sum(combination[target - nums[i]]) target>=nums[i]

1: when target=0,

}

## **Recursive implementation similar to Coin Change Problem <https://leetcode.com/problems/coin-change-2/>**

/\*\*

\* Recurrence relation

\* \* comb[target] = {

\* \* sum(comb[target - nums[i]]) target> 0

\* \* 1: when target=0,

\* \* }

\* <p>

\* Complexity: O(n\*2^n)

\*/

static class CombinationSumIVRecursive {

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

if (nums == null || nums.length == 0)

return 0;

//base case

if (target == 0)

return 1;

int res = 0;

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

if (target >= nums[i])

res += combinationSum4(nums, target - nums[i]);

return res;

}

}

**Top-Down:** Many sub-problem, cache it

/\*\*

\* Recurrence relation

\* \* comb[target] = {

\* \* sum(comb[target - nums[i]]) target> 0

\* \* 1: when target=0,

\* \* }

\* Overlapping sub-problems; cache it

\* Complexity: O(n\*n)

\*/

static class CombinationSumIVTopDown {

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

if (nums == null || nums.length == 0)

return 0;

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

Arrays.fill(dp, -1);

return combinationSum4(nums, target, dp);

}

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

if (target < 0)

return 0;

//base case

if (target == 0)

return 1;

if (dp[target] != -1)

return dp[target];

int res = 0;

for (int i = 0; i < nums.length; i++) //O(n)

if (target >= nums[i])

res += combinationSum4(nums, target - nums[i]); //O(n)

return dp[target] = res;

}

}

**Bottom Up :**

After above, i realized that it should be fastest. But you see its just 84.17% beat.

Then, i thought instead of doing reverse (target -> 0), doing forward (sum -> target) would be fastest.

As sum grows towards target faster than target shrink towards 0.

/\*\*

\* Recurrence relation

\* \* comb[target] = {

\* \* sum(comb[target - nums[i]]) target> 0

\* \* 1: when target=0,

\* \* }

\* Overlapping sub-problems; cache it

\* Complexity: O(n\*n)

\* <p>

\* Runtime: 0 ms, faster than 100.00% of Java online submissions for Combination Sum IV.

\* Memory Usage: 34.2 MB, less than 100.00% of Java online submissions for Combination Sum IV.

\*/

static class CombinationSumIVTopDownV2 {

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

if (nums == null || nums.length == 0)

return 0;

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

Arrays.fill(dp, -1);

return combinationSum4(nums, 0, target, dp);

}

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

if (currSum > target)

return 0;

//base case

if (target == currSum)

return 1;

if (dp[currSum] != -1)

return dp[currSum];

int res = 0;

for (int i = 0; i < nums.length; i++) //O(n)

res += combinationSum4(nums, currSum + nums[i], target, dp); //O(n)

return dp[currSum] = res;

}

}

**Solution 3: Pick or No-pick [0-1 knapsack] (60 min)**

**Similar as L39.Combination Sum Backtracking style 2, TLE**

class Solution {

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

if(candidates == null || candidates.length == 0) {

return 0;

}

return helper(candidates, target, 0);

}

private int helper(int[] candidates, int target, int index) {

if(index == candidates.length || target < 0) {

return 0;

}

if(target == 0) {

return 1;

}

// The different part than L39.Combination Sum is always start

// from 0 instead of 'index', since order matters and L377

// is permutation internally

int pick = helper(candidates, target - candidates[index], 0);

int no\_pick = helper(candidates, target, index + 1);

return pick + no\_pick;

}

}

**Adding 2D memo as Top Down DP**

class Solution {

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

if(candidates == null || candidates.length == 0) {

return 0;

}

Integer[][] memo = new Integer[candidates.length + 1][target + 1];

return helper(candidates, target, 0, memo);

}

private int helper(int[] candidates, int target, int index, Integer[][] memo) {

if(index == candidates.length || target < 0) {

return 0;

}

if(target == 0) {

return 1;

}

if(memo[index][target] != null) {

return memo[index][target];

}

// The different part than L39.Combination Sum is always start

// from 0 instead of 'index', since order matters and L377

// is permutation internally

int pick = helper(candidates, target - candidates[index], 0, memo);

int no\_pick = helper(candidates, target, index + 1, memo);

memo[index][target] = pick + no\_pick;

return pick + no\_pick;

}

}

**Refer to**

<https://leetcode.com/problems/combination-sum-iv/solutions/111860/coin-change-and-this-problem/comments/887001>

From a memoization point of view. Only difference is that permutations have to start from 0 (to take other rearrangements).

Permutations can pass with 1d cache. Combinations must have a 2d cache (\*For this kind of memo approach)

class Solution { // coin change 2

public int change(int amount, int[] coins) {

return count(coins, amount, 0, new Integer[coins.length][amount + 1]);

}

private int count(int[] nums, int target, int pos, Integer[][] cache) {

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

return (target == 0) ? 1 : 0;

}

if (cache[pos][target] != null) {

return cache[pos][target];

}

int take = count(nums, target - nums[pos], pos, cache);

int skip = count(nums, target, pos + 1, cache);

return cache[pos][target] = take + skip;

}

}

This problem, for take branch the 'pos' change to '0'

class Solution { // this problem

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

return count(nums, target, 0, new Integer[target + 1]);

}

private int count(int[] nums, int target, int pos, Integer[] cache) {

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

return (target == 0) ? 1 : 0;

}

if (cache[target] != null) {

return cache[target];

}

int take = count(nums, target - nums[pos], 0, cache);

int skip = count(nums, target, pos + 1, cache);

return cache[target] = take + skip;

}

}

<https://leetcode.com/problems/combination-sum-iv/solutions/1701806/comparing-coin-change-2-and-combination-sum-4/>

First thing we need to understand is what is permutation and combination ?In permutations - (1,1,2) ,(1,2,1),(2,,1,1) are treated as different and counted where as in combinations all these are considered same. The question is combination sum 4 but we need to print the permutation count

Now coming to the coin change 2 and this problem

In coin change 2 we are using combinations and combination sum 4 we are using permutations coin change 2 solution

class Solution {

public int change(int amount, int[] coins) {

return fn(amount,coins,0,new HashMap<String,Integer>());

}

int fn(int amount,int[] coins,int ci,HashMap<String,Integer> memo)

{

if(ci>=coins.length||amount<=0)

return (amount==0)?1:0;

String currentKey = Integer.toString(ci)+'\_'+Integer.toString(amount);

if(memo.containsKey(currentKey))

return memo.get(currentKey);

int consider = fn(amount-coins[ci],coins,ci,memo);

int notConsider = fn(amount,coins,ci+1,memo);

memo.put(currentKey,consider+notConsider);

return consider+notConsider;

}

}

Now coming to combination sum 4 code

class Solution {

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

return fn(nums,target,0,new HashMap<Integer,Integer>());

}

public int fn(int[] nums,int target,int ci,HashMap<Integer,Integer> memo)

{

if(ci>=nums.length||target<=0)

return (target==0)?1:0;

int currentKey = target;

if(memo.containsKey(currentKey))

return memo.get(currentKey);

int consider = fn(nums,target-nums[ci],0,memo);

int notConsider = fn(nums,target,ci+1,memo);

memo.put(currentKey,consider+notConsider);

return consider+notConsider;

}

}

The difference is in coin change 2 , currentKey is dependent on two values(2d DP) on currentIndex and amount

whereas in combinationSum 4 our currentKey is dependent on only target(1d DP) and in the consider call we are passing zero instead of currentIndex

Let me explain you this with an example

[1,2,5] target = 5

In coin change 2 the possible outcomes are

[1,1,1,1,1]

[1,2,2]

[1,1,1,2]

[5]

In the above case the output is 4 (4 ways to make the required amount as 5)

Now in combination sum 4 the possible outcomes are

[1,1,1,1,1]

[1,2,2]

[2,2,1]

[1,1,1,2]

[1,2,1,1]

[1,1,2,1]

[2,1,1,1]

[5]

These are the 9 cases

Now if you clearly once we selected 1 and moved to next index 2 as in combination [1,2,2], nums =[1,2,5], index =0 1 2

in this case at the 0 index we didn't consider and moved to index 1 that is element 2

again we are coming back to index and considering the element 1 so the combination [2,2,1]

If we start from currentIndex (ci) then we are always moving ahead but here we need to count all the rearrangements

that's why we are including 0 instead of currentIndex