<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:**

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

**Style 1: start with i = 0 and pass in 'i' (TLE 10/16)**

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

**Style 2: start with i = index and pass in 'index', which actually similar to Leetcode** [L322.Coin Change (Ref.L39,L46,L377)](note://WEB99a663d0191c2bc0c88aec7cf5f3d9fd)**, and NOT Leetcode** [L39.Combination Sum (Ref.L40,L322)](note://79A77D14319840B9A134C272EA618600)**, since the substance / essense of L377 is a permutation rather than a combination since the order matters. (TLE 10/16)**

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 = 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];

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

                target += candidates[i];

            }

        }

        return res;

    }

}

Time Complexity : O(2^N)

Space Complexity: O(2^N)

**Why for(int i = index; i < ...) & pass 'i' in each recursion level is wrong in L377.Combination Sum IV(but in L39.Combination Sum it was correct)?**

**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 Deepseek**

**Why we can remove "backtracking" logic in Solution 2 (compare with Solution 1's style 1) ? is that related to kind of Java nature, something like "pass by value" in recursive call ?**

Okay, let's try to figure out why the backtracking logic can be removed in the second solution for LeetCode 377. The user mentioned Java's "pass by value" nature, so maybe that plays a role here.

First, I need to recall what pass by value means in Java. I remember that Java passes primitive data types (like int) by value. So when you pass an int variable to a method, a copy of its value is made. Any changes to that variable inside the method don't affect the original variable outside. Objects, on the other hand, are passed by reference, but for primitives, it's strictly pass by value.

Looking at Solution 1, they modify the target variable directly:

target -= candidates[i];

res += combinationSum4(candidates, target);

target += candidates[i];

Here, they subtract the candidate from the target, make the recursive call, and then add it back. That's classic backtracking—modifying a state, recursing, and then undoing the modification to explore other paths.

In Solution 2, they don't modify the target. Instead, they pass the updated value directly to the helper function:

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

Since target is a primitive (int), when they call helper with target - candidates[i], they're passing a new value. The original target variable in the current method call remains unchanged. This means there's no need to backtrack because each recursive call works with its own copy of the target value. Each stack frame has its own target, so changing it in one call doesn't affect others.

So, the key here is that Java passes primitives by value. In Solution 1, modifying target and then restoring it is necessary because they're altering the same variable in the current stack frame. But in Solution 2, since each recursive call gets a new value (target - candidate), there's no shared state to modify or restore. Hence, backtracking isn't needed.

This makes sense. By avoiding state mutation and instead passing new values, Solution 2 eliminates the need for backtracking steps. Each recursive call operates on its own target value, so there's no interference between different branches of the recursion tree. This approach is cleaner and less error-prone because you don't have to remember to undo changes.

Wait, but why does Solution 1 need to backtrack if Java passes by value? Oh, wait—in Solution 1, they are modifying the target variable in the loop. Since they're reusing the same variable for each iteration, after the recursive call returns, they need to add the candidate back to target to restore it for the next iteration. Even though each recursive call gets a copy, the loop in the current call is reusing the same target variable. So without backtracking, the target would be incorrect for subsequent iterations in the same loop.

In contrast, Solution 2 doesn't modify the target variable at all. They calculate target - candidates[i] and pass that directly, so the original target remains the same for all iterations. There's no need to adjust it back because it's never changed in the first place.

So the difference is that Solution 1 modifies the state (target) during iteration and needs to backtrack to ensure the correct state for each loop iteration. Solution 2 avoids modifying the state by passing the new value directly, leveraging Java's pass-by-value for primitives. This eliminates the need for backtracking because each recursive call works with its own independent value of target.

Therefore, the ability to remove backtracking in Solution 2 is indeed related to Java's pass-by-value mechanism. Since each recursive call receives a copy of the target value, there's no shared state that needs to be reset after the call returns. Each iteration of the loop uses the original target value, ensuring correctness without manual backtracking.

The ability to remove backtracking logic in the second solution for LeetCode 377 (Combination Sum IV) is directly tied to Java's "pass-by-value" mechanism and how recursive calls handle primitive variables like int. Here's a clear breakdown:

**Why Backtracking is Removed in Solution 2:**

Java Passes Primitives by Value:

When you pass an int (e.g., target) to a method, a copy of its value is created.

Modifying the copy inside the method does not affect the original variable in the caller.

Solution 1 Modifies target:

target -= candidates[i]; // Modify target

res += combinationSum4(candidates, target); // Recurse with modified target

target += candidates[i]; // Backtrack: restore target for next iteration

Here, target is modified directly, and backtracking (target += candidates[i]) is required to restore it for subsequent loop iterations.

Without backtracking, the modified target would corrupt the state for other candidates in the loop.

Solution 2 Avoids State Mutation:

res += helper(candidates, target - candidates[i]); // Pass a new value, no mutation

Instead of modifying target, the updated value (target - candidates[i]) is passed directly to the recursive call.

The original target remains unchanged for all iterations of the loop, eliminating the need for backtracking.

**Key Insight:**

Solution 1 uses mutable state (modifying target), requiring explicit backtracking to reset the state after recursion.

Solution 2 uses immutable state (passing a new value), leveraging Java’s pass-by-value to ensure each recursive call works with its own copy of target.

No shared state between recursive calls → No need to backtrack.

**Example Walkthrough:**

For candidates = [1, 2], target = 3:

**Solution 1 (With Backtracking):**

Loop Iteration 1 (i=0, candidate 1):

target = 3 - 1 = 2 (modify).

Recurse with target=2.

After recursion, restore target = 2 + 1 = 3.

Loop Iteration 2 (i=1, candidate 2):

target = 3 - 2 = 1 (modify).

Recurse with target=1.

After recursion, restore target = 1 + 2 = 3.

Backtracking ensures target is restored for each iteration.

**Solution 2 (No Backtracking):**

Loop Iteration 1 (i=0, candidate 1):

Pass target - 1 = 2 to recursion.

Original target = 3 remains unchanged.

Loop Iteration 2 (i=1, candidate 2):

Pass target - 2 = 1 to recursion.

Original target = 3 is still intact.

No state mutation → No restoration needed.

**Why This Works:**

Pass-by-Value: Each recursive call operates on a new copy of target, leaving the original target in the caller untouched.

Immutability: By avoiding state changes, the code naturally avoids side effects between loop iterations.

**Conclusion:**

Solution 2 removes backtracking by leveraging Java’s pass-by-value for primitives.

By passing immutable values (target - candidates[i]) instead of mutating target, the code becomes simpler and avoids manual state management.

This approach aligns with functional programming principles, where immutability reduces complexity and errors.

**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;

    }

}

**Refer to Deepseek**

**For Leetcode 377, on pure DFS solutions, besides above for loop (both backtrack and non-backtrack use for loop) style, i have seen another divide and conquer style (maybe a 0-1 knapsack ?)**

Okay, let's take a look at this problem. The user is asking about the differences between two approaches to solve LeetCode 377 (Combination Sum IV): one using a for loop and backtracking, and another using a divide and conquer (or 0-1 knapsack style) approach. They want to know how these solutions compare and when to use each.

First, I need to understand both solutions thoroughly. The for-loop approach, as discussed earlier, explores all possible permutations by iterating through each candidate and recursively subtracting the current candidate from the target. The divide and conquer solution here splits the problem into two parts: picking the current element (and allowing reuse by resetting the index to 0) and not picking it, then moving to the next element.

Let me start by analyzing the divide and conquer code provided. The helper function takes the candidates array, target, and an index. The base cases check if the index is out of bounds or the target is negative (return 0), or if the target is zero (return 1). Then, it splits into two recursive calls: "pick" and "no\_pick". The pick call subtracts the current candidate from the target and resets the index to 0, allowing the same element to be picked again but in different orders. The no\_pick call moves to the next index without using the current candidate.

Comparing this to the for-loop solutions, the key difference is how they handle the exploration of permutations. The for-loop approach explicitly iterates through each candidate at every step, leading to a natural exploration of all permutations. The divide and conquer method, on the other hand, structures the recursion by deciding at each step whether to include the current candidate (with a reset index) or exclude it, which also allows permutations but in a different way.

Now, why would one use the divide and conquer approach over the for-loop? The user also asks when the divide and conquer method is necessary. I need to think about the scenarios where each approach is more suitable. For instance, if the problem constraints or structure naturally fit a decision tree where each step involves including or excluding an element, the divide and conquer might be more intuitive. However, in terms of efficiency, both approaches without memoization would have exponential time complexity, but the for-loop might have more redundant checks due to the loop structure.

Another point to consider is how each approach handles the order of elements. Since Combination Sum IV counts permutations, the reset of the index in the pick call (to 0) ensures that previous elements can be reused in subsequent steps, which is essential for generating all permutations. The for-loop approach achieves the same by starting the loop from 0 each time, but the divide and conquer does it through the recursive structure.

I should also think about the potential for memoization or dynamic programming optimizations. The divide and conquer approach might be easier to memoize because the state is determined by the target and the index, whereas the for-loop's state is just the target. However, in the given code examples, neither uses memoization, so both are pure DFS solutions.

In terms of code structure, the for-loop is more straightforward for generating permutations because it directly iterates through all possible candidates at each step. The divide and conquer approach requires a mental model of including/excluding elements and resetting the index, which might be less intuitive for some but aligns with classic recursive problem-solving techniques.

To summarize the comparison: both approaches correctly count permutations, but they structure the recursion differently. The for-loop is more explicit in exploring all candidates at each step, while the divide and conquer uses a decision-based split. The choice between them might depend on the problem's specific requirements or personal coding style, but in terms of correctness for LeetCode 377, both work. However, considering efficiency and potential optimizations, the for-loop might be more naturally extended with memoization by tracking the target alone, whereas the divide and conquer would require tracking both target and index.

Finally, when must one use the divide and conquer approach? If the problem requires making a series of decisions (include/exclude) and naturally fits a binary split at each step, then the divide and conquer is appropriate. However, for permutation problems where order matters and elements can be reused, the for-loop approach might be more direct unless the problem's constraints or structure align better with the divide and conquer's decision tree.

The divide-and-conquer (DAC) style solution and the for-loop-based DFS solution for LeetCode 377 (Combination Sum IV) both count valid permutations, but they differ in their recursive structure, state management, and applicability. Here's a detailed comparison:

**Key Differences:**

|  |  |  |
| --- | --- | --- |
| Feature | For-Loop DFS (Previous Solutions) | Divide-and-Conquer (DAC) Solution |
| **Recursive Structure** | Iterates through all candidates at each step. | Splits into two decisions: "pick" or "skip" the current candidate. |
| **Permutations Handling** | Explicitly explores permutations via loops. | Implicitly allows permutations by resetting index to 0 when picking. |
| **State Parameters** | Relies on target and loop iteration. | Uses target and index  (current candidate position). |
| **Redundancy** | Explores all permutations directly. | May recompute overlapping subproblems (e.g., same target  with different index). |
| **Memoization Friendliness** | Harder to memoize (state depends only on target). | Easier to memoize (state depends on target and index). |

**Why the DAC Solution Works for LeetCode 377:**

Decision Logic:

Pick: Subtract candidates[index] from target and reset index to 0 to allow permutations (earlier candidates can be reused in subsequent steps).

Skip: Move to the next candidate (index + 1) without using the current one.

Permutations via Index Reset:

By resetting index to 0 in the "pick" branch, the DAC solution ensures that earlier candidates can be reused in any order.

Example: For candidates = [1, 2] and target = 3, the "pick" path allows sequences like [1, 2] and [2, 1].

**When to Use the DAC Approach:**

Decision-Oriented Problems:

Use DAC when the problem requires binary decisions (e.g., include/exclude an element, as in the 0-1 knapsack problem).

Example: Counting subsets or partitions where order doesn’t matter.

Memoization Opportunities:

The DAC approach’s state (target and index) is easier to memoize, avoiding redundant subproblem computations.

Example: Adding memoization to the DAC solution reduces time complexity from exponential to polynomial.

Explicit Control Over Element Reuse:

The DAC approach clearly separates "reusing" an element (via index reset) from "skipping" it.

**Comparison with For-Loop DFS:**

|  |  |  |
| --- | --- | --- |
| Aspect | For-Loop DFS | DAC Solution |
| **Permutations Generation** | Explicit via loop iteration. | Implicit via index reset in "pick" branch. |
| **Code Readability** | Intuitive for permutation problems. | Requires understanding of decision splits. |
| **Memoization Potential** | Harder (state = target). | Easier (state = target + index). |
| **Redundancy** | High (recomputes same target  in loops). | Moderate (recomputes same  target + index). |

**When to Prefer DAC:**

Memoization is Required:

DAC’s state (target and index) is ideal for memoization. For example, using memo[target][index] avoids redundant calls.

For-loop DFS would require memoizing only target, which may conflate states and undercount permutations.

Decision-Based Problems:

If the problem involves explicit choices (e.g., "use element or not"), DAC aligns naturally.

Example: LeetCode 40 (Combination Sum II), where duplicates must be skipped.

Avoiding Loops:

DAC avoids nested loops, which can simplify code in some cases.

**Why For-Loop DFS is Often Better for LeetCode 377:**

Direct Permutation Handling: The loop structure explicitly iterates through all candidates at each step, naturally counting permutations.

Simplicity: No need to manage index parameters or decision splits.

**Conclusion:**

For-Loop DFS is more straightforward for LeetCode 377, as it directly aligns with the problem’s permutation requirements.

DAC is useful when decisions (pick/skip) need explicit control, or when memoization is required.

Use DAC when:

The problem requires binary decisions (e.g., 0-1 knapsack).

Memoization is needed for optimization.

Element reuse logic is complex (e.g., resetting index for permutations).

Both approaches are valid, but the choice depends on the problem’s constraints and the need for optimization. For pure DFS without memoization, the for-loop solution is simpler. For optimized DP, DAC with memoization is better.

**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

**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 (Ref.L377)](note://WEB9160d308503a5b8c326712331320ec56) **and L377 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, 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);

    }

**Refer to**

[L39.Combination Sum (Ref.L40,L322)](note://79A77D14319840B9A134C272EA618600)

[L40.Combination Sum II (Ref.L39.L90)](note://7895D4AD64EC43CA903579F3C8F803AE)

[L322.Coin Change (Ref.L39,L46,L377)](note://WEB99a663d0191c2bc0c88aec7cf5f3d9fd)

[L518.Coin Change II (Ref.L377)](note://WEB9160d308503a5b8c326712331320ec56)