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

Given a collection of candidate numbers (candidates) and a target number (target), find all **unique** combinations in candidates where the candidate numbers sum to target.

Each number in candidates may only be used **once** in the combination.

**Note:** The solution set must not contain duplicate combinations.

**Example 1:**

**Input:** candidates = [10,1,2,7,6,1,5], target = 8

**Output:** [[1,1,6],[1,2,5],[1,7],[2,6]]

**Example 2:**

**Input:** candidates = [2,5,2,1,2], target = 5

**Output:** [[1,2,2],[5]]

**Constraints:**

1 <= candidates.length <= 100

1 <= candidates[i] <= 50

1 <= target <= 30

**Attempt 1: 2022-10-13**

**Solution 1: Backtracking style 1 (10min)**

class Solution {

    public List<List<Integer>> combinationSum2(int[] candidates, int target) {

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

        Arrays.sort(candidates);

        helper(target, candidates, result, new ArrayList<Integer>(), 0);

        return result;

    }

    private void helper(int target, int[] candidates, List<List<Integer>> result, List<Integer> tmp, int index) {

        if(target == 0) {

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

            return;

        }

        if(target < 0) {

            return;

        }

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

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

                continue;

            }

            tmp.add(candidates[i]);

            helper(target - candidates[i], candidates, result, tmp, i + 1);

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

        }

    }

}

Complexity analysis:

https://leetcode.com/problems/combination-sum/discuss/1755084/Detailed-Time-and-Space-Complecity-analysisc++javabacktracking

Time Complexity: O(nlogn + 2^n) ~= O(2^n) where n is size of candidates array

If we were not allowed to pick a single element multiple times than the time complexity would be 2^n

(where n is the size of candidates array), because we will only have 2 choices per element

e.g    [2,    3,    6,    7]

        /\    /\    /\    /\

      t2 nt2 t3 nt3 t6 nt6 t7 nt7

        2  \*  2  \*  2  \*  2 --> 2^4 (4 is size of candidate array)

Also the nlogn because deep copy of array in each recursion level will cost logn time and candidate array length is n,

so total deep copy time is nlogn, but compare to exponentially increase 2^n part, nlogn can be ignored.

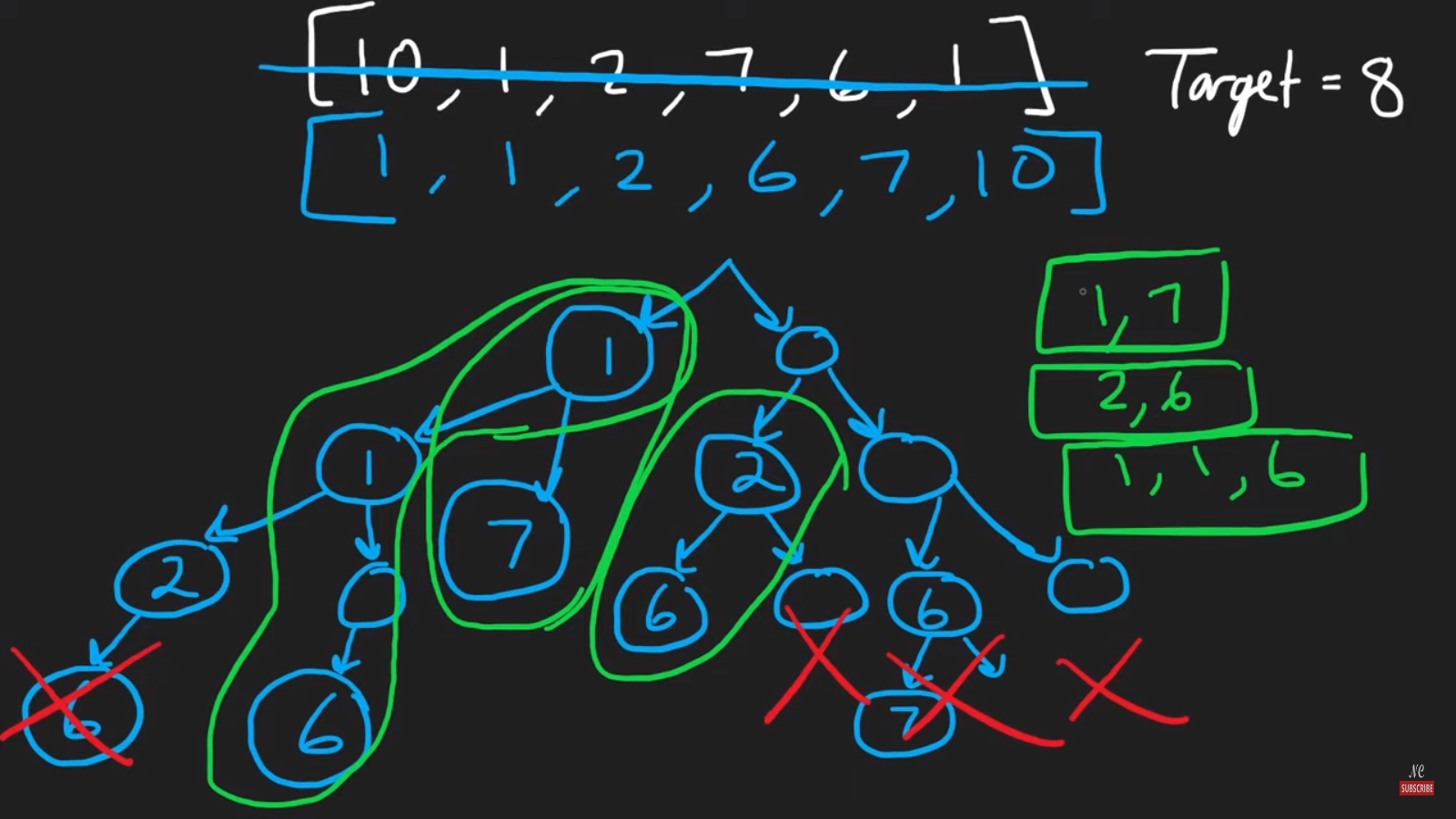
Space Complexity: O(length\_of\_longest\_combination)

**Video explain why and how to skip duplicate elements only on particular "Not pick" branch**

[Combination Sum II - Backtracking - Leetcode 40 - Python](https://www.youtube.com/watch?v=rSA3t6BDDwg)

<https://www.youtube.com/watch?v=rSA3t6BDDwg>

**Example for candidates={10,1,2,7,6,1}, target=8, the for loop backtracking style 1 final status as below:**

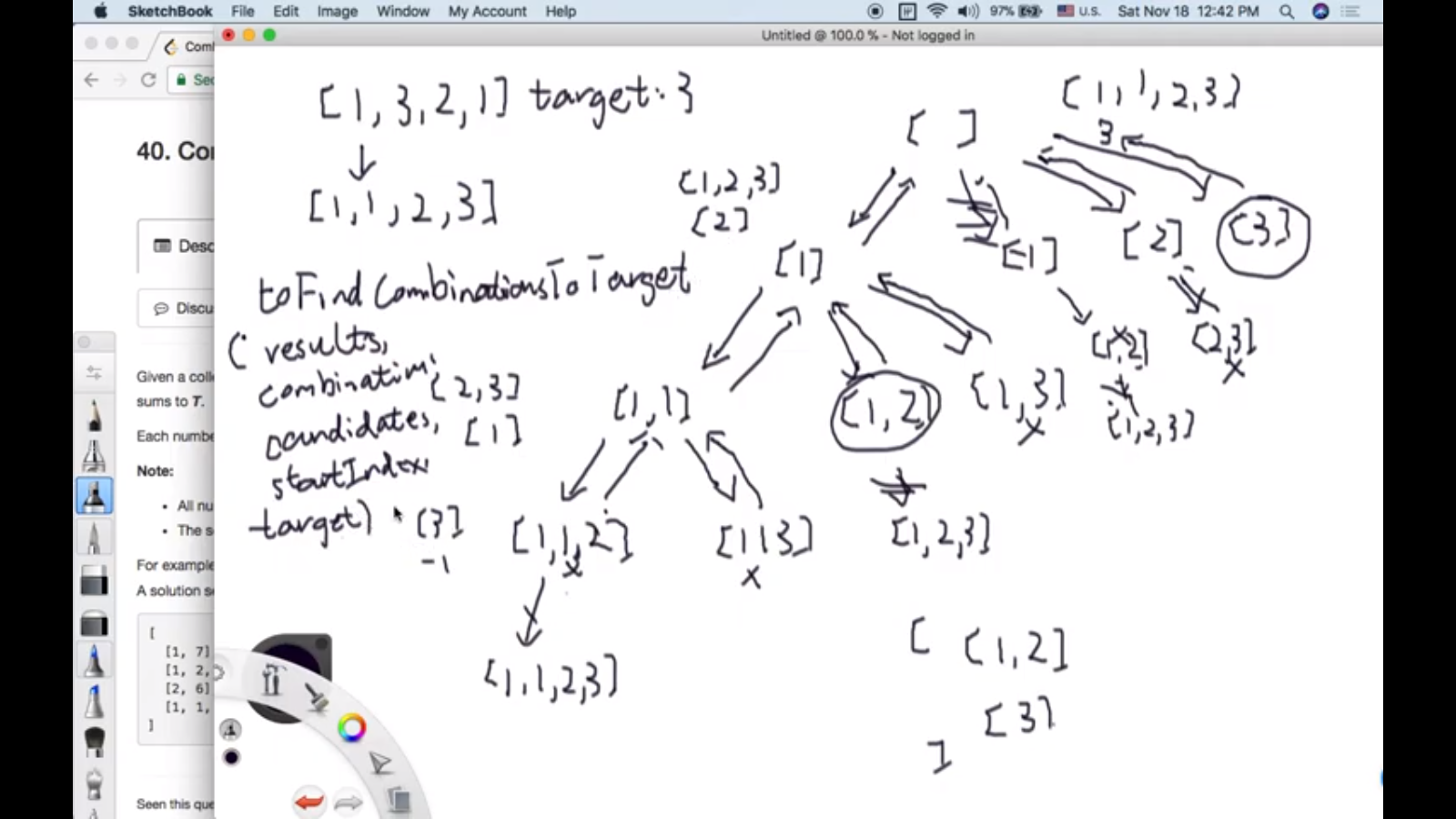


**Another video explain**

[LeetCode Tutorial 40. Combination Sum II](https://www.youtube.com/watch?v=j9_qWJClp64)

<https://www.youtube.com/watch?v=j9_qWJClp64>

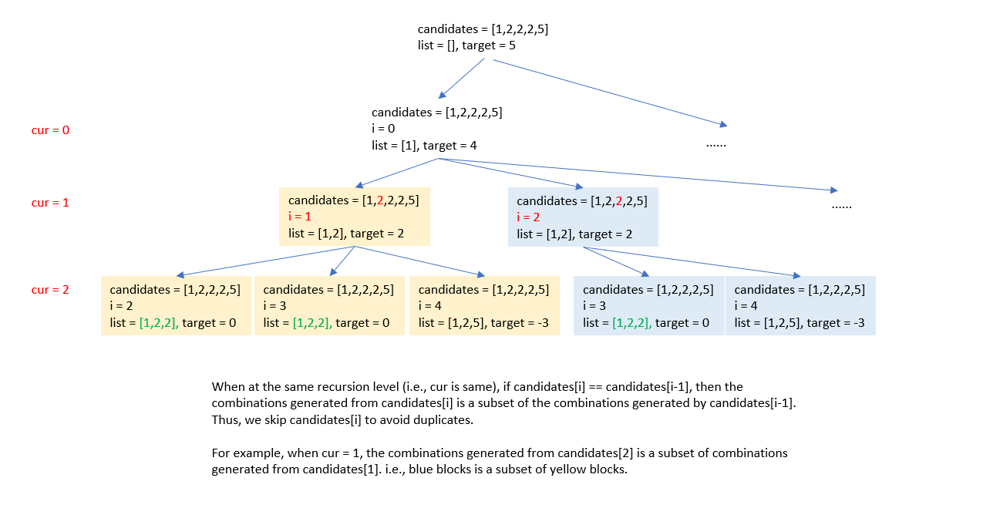
**Example for candidates={1,3,2,1}, target=3, the for loop backtracking style 1 final status as below:**



**What's the meaning of**

if (i > cur && candidates[i] == candidates[i-1]) continue;**?**

<https://leetcode.com/problems/combination-sum-ii/discuss/16861/Java-solution-using-dfs-easy-understand/977097>



**Solution 2: Backtracking style 2 (10min)**

**Keep the same style as L90/P11.2 Subsets II to skip duplicate elements only on particular "Not pick" branch**

**For why skip only on "Not pick" branch, refer L90/P11.2 Subsets II**

**Video explain why and how to skip duplicate elements only on particular "Not pick" branch**

[Subsets II - Backtracking - Leetcode 90 - Python](https://www.youtube.com/watch?v=Vn2v6ajA7U0)

<https://www.youtube.com/watch?v=Vn2v6ajA7U0>

**Correct solution 2.1 with local variable 'i' to skip duplicate elements on particular "Not pick" branch, since go with "Not pick" branch before "Pick" branch, if no separate variable 'i' to isolate  skip only on "Not pick" branch and not on "Pick" branch, skip will impact "Pick" branch accidently**

class Solution {

    public List<List<Integer>> combinationSum2(int[] candidates, int target) {

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

        Arrays.sort(candidates);

        helper(target, candidates, result, new ArrayList<Integer>(), 0);

        return result;

    }

    private void helper(int target, int[] candidates, List<List<Integer>> result, List<Integer> tmp, int index) {

        if(target == 0) {

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

            return;

        }

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

            return;

        }

        // Not pick

        int i = index;

        while(i + 1 < candidates.length && candidates[i] == candidates[i + 1]) {

            i++;

        }

        helper(target, candidates, result, tmp, i + 1);

        // Pick

        tmp.add(candidates[index]);

        helper(target - candidates[index], candidates, result, tmp, index + 1);

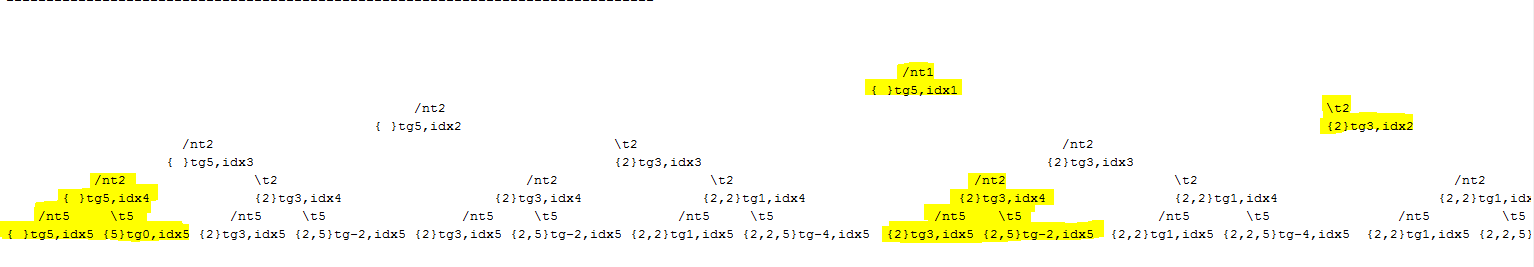
        tmp.remove(tmp.size() - 1);

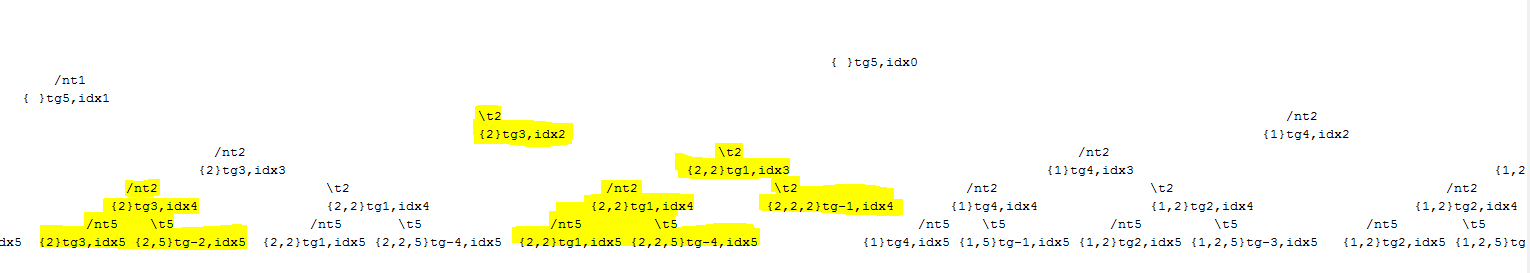
    }

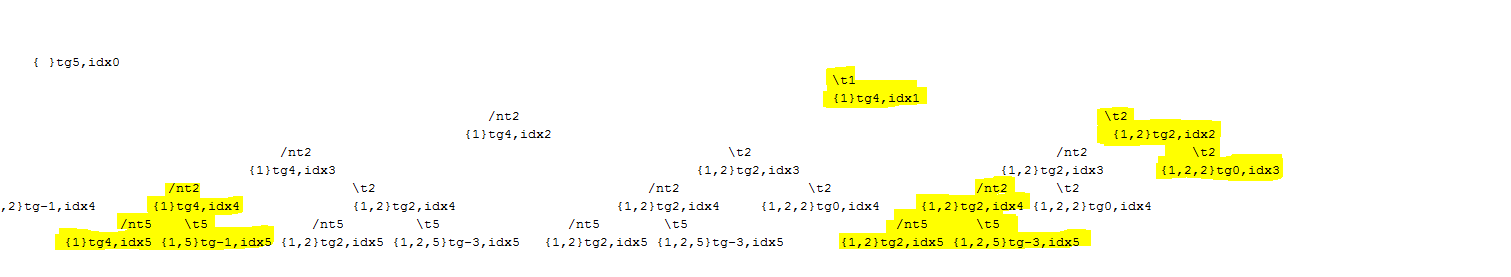
}

**How the skip duplicate elements  on "Not pick" branch prune branches ?**

**Note: Yellow highlighted branches are the remain branches, all other branches pruned**







**Above snapshots based on not skip duplicate elements Backtracking style 2 recursion step by step (to view stretch on Notepad++)**

candidates={1,2,2,2,5},target=5(tg5),index=0(idx0)

branch ending condition: (1)index >=5 -> idx5 OR (2)target < 0 -> tg(<0)

combination found: target == 0(tg0)

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

What will happen if not skip duplicate elements Backtracking style 2 recursion step by step (to view stretch on Notepad++) ?

(1) tg0 -> one {5} on not take 1 branch, three {1,2,2} on take 1 branch

(2) Both not take and take certain element branch needs skip duplicates as we can see duplicate subtrees happen on both not take and take branch, so no need to create local variable 'i' to isolate skip duplicates process only on not take element branch like L90 Backtracking style 2

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

                                                                                                                                                                                                                { }tg5,idx0

                                                                                                                /nt1                                                                                                                                                                                                \t1

                                                                                                            { }tg5,idx1                                                                                                                                                                                              {1}tg4,idx1

                                                  /nt2                                                                                                              \t2                                                                                                  /nt2                                                                        \t2

                                              { }tg5,idx2                                                                                                            {2}tg3,idx2                                                                                      {1}tg4,idx2                                                                      {1,2}tg2,idx2

                      /nt2                                                  \t2                                                    /nt2                                                          \t2                                          /nt2                                                    \t2                                      /nt2            \t2

                    { }tg5,idx3                                            {2}tg3,idx3                                          {2}tg3,idx3                                                  {2,2}tg1,idx3                                {1}tg4,idx3                                            {1,2}tg2,idx3                        {1,2}tg2,idx3      {1,2,2}tg0,idx3

          /nt2                \t2                              /nt2                  \t2                              /nt2                    \t2                                /nt2                \t2                    /nt2                  \t2                                  /nt2                \t2                  /nt2      \t2

      { }tg5,idx4            {2}tg3,idx4                  {2}tg3,idx4              {2,2}tg1,idx4                  {2}tg3,idx4                {2,2}tg1,idx4                    {2,2}tg1,idx4        {2,2,2}tg-1,idx4      {1}tg4,idx4              {1,2}tg2,idx4                    {1,2}tg2,idx4    {1,2,2}tg0,idx4    {1,2}tg2,idx4 {1,2,2}tg0,idx4

    /nt5    \t5            /nt5    \t5                /nt5    \t5              /nt5    \t5                    /nt5    \t5                /nt5      \t5                /nt5        \t5                              /nt5    \t5            /nt5        \t5                /nt5        \t5                        /nt5      \t5

{ }tg5,idx5{5}tg0,idx5{2}tg3,idx5{2,5}tg-2,idx5 {2}tg3,idx5 {2,5}tg-2,idx5 {2,2}tg1,idx5 {2,2,5}tg-4,idx5  {2}tg3,idx5 {2,5}tg-2,idx5  {2,2}tg1,idx5 {2,2,5}tg-4,idx5  {2,2}tg1,idx5{2,2,5}tg-4,idx5            {1}tg4,idx5 {1,5}tg-1,idx5{1,2}tg2,idx5 {1,2,5}tg-3,idx5  {1,2}tg2,idx5 {1,2,5}tg-3,idx5      {1,2}tg2,idx5 {1,2,5}tg-3,idx5

**Refer to**

<https://leetcode.com/problems/combination-sum-ii/discuss/1671782/C%2B%2B-100-backtrack>

class Solution {

public:

    vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {

        vector<vector<int>> ans;

        vector<int> curr;

        sort(candidates.begin(), candidates.end());

        backtrack(candidates, target, ans, curr, 0);

        return ans;

    }

        if(target == 0) {

            ans.push\_back(curr);

            return;

        }

        if(index >= candidates.size()) return;

        if(target >= candidates[index]) {

            curr.push\_back(candidates[index]);

            backtrack(candidates, target - candidates[index], ans, curr, index + 1);

            curr.pop\_back();

            while(index + 1 < candidates.size() && candidates[index + 1] == candidates[index]) ++index;

            backtrack(candidates, target, ans, curr, index + 1);

        }

    }

};

**Correct solution 2.2, same as 2.1, different style on how to skip duplicate elements**

class Solution {

    public List<List<Integer>> combinationSum2(int[] candidates, int target) {

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

        Arrays.sort(candidates);

        helper(target, candidates, result, new ArrayList<Integer>(), 0);

        return result;

    }

    private void helper(int target, int[] candidates, List<List<Integer>> result, List<Integer> tmp, int index) {

        if(target == 0) {

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

            return;

        }

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

            return;

        }

        // Not pick

        int i = index;

        while(i < candidates.length && candidates[i] == candidates[index]) {

            i++;

        }

        helper(target, candidates, result, tmp, i);

        // Pick

        tmp.add(candidates[index]);

        helper(target - candidates[index], candidates, result, tmp, index + 1);

        tmp.remove(tmp.size() - 1);

    }

}

**Refer to**

<https://leetcode.com/problems/subsets-ii/discuss/169226/Java-Two-Way-of-Recursive-thinking>

class Solution {

    public List<List<Integer>> subsetsWithDup(int[] nums) {

        Arrays.sort(nums);

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

        helper(res,new ArrayList<>(),nums,0,false);

        return res;

    }

    public void helper(List<List<Integer>> res, List<Integer> ls, int[] nums, int pos, boolean choosePre) {

        if(pos==nums.length) {

            res.add(new ArrayList<>(ls));

            return;

        }

        helper(res,ls,nums,pos+1,false);

        if(pos>=1&&nums[pos]==nums[pos-1]&&!choosePre) return;

        ls.add(nums[pos]);

        helper(res,ls,nums,pos+1,true);

        ls.remove(ls.size()-1);

    }

}

**Different styles to skip duplicate elements in correct solution 2.1 and 2.2?**

e.g

For sorted array nums={1,2,2,2,5}, index=1, all duplicate '2' stored continuously in array

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

For solution 2.1

int i = index;

while(i < nums.length && nums[i] == nums[index]) {i++;}

helper(nums, result, tmp, i);

=> while loop ending when i=4, nums[4]=5 != nums[1]=2, not pick up branch skip all duplicate 2 and start from 5 requires pass i(=4) to next recursion

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

For solution 2.2

int i = index;

while(i + 1 < nums.length && nums[i] == nums[i + 1]) {i++;}

helper(nums, result, tmp, i + 1);

=> while loop ending when i=3, nums[3]=2 != nums[4]=5, not pick up branch skip all duplicate 2 and start from 5 requires pass i + 1(=4) to next recursion

**Also switch order of "Not pick" and "Pick" doesn't matter when have a local variable 'i' to handle skip duplicate elements on "Not pick" branch only, but it matters if no local variable 'i',  then "Pick" branch must comes before "Not pick" branch because if not comes this order, "Not pick" branch update 'index' will impact "Pick" branch who comes later, without local variable 'i' solution refer to 2.3**

class Solution {

    public List<List<Integer>> combinationSum2(int[] candidates, int target) {

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

        Arrays.sort(candidates);

        helper(target, candidates, result, new ArrayList<Integer>(), 0);

        return result;

    }

    private void helper(int target, int[] candidates, List<List<Integer>> result, List<Integer> tmp, int index) {

        if(target == 0) {

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

            return;

        }

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

            return;

        }

        // If have a local variable 'i' to handle skip duplicate elements on "Not pick" branch only,

        // we can switch order of "Pick" and "Not pick" branch

        // Pick

        tmp.add(candidates[index]);

        helper(target - candidates[index], candidates, result, tmp, index + 1);

        tmp.remove(tmp.size() - 1);

        // Not pick

        int i = index;

        while(i < candidates.length && candidates[i] == candidates[index]) {

            i++;

        }

        helper(target, candidates, result, tmp, i);

    }

}

**Correct solution 2.3 without local variable 'i' to skip duplicate elements on particular "Not pick" branch, since go with "Pick" branch before "Not pick" branch, even no separate variable 'i' to isolate, skip will only impact "Not pick" branch**

class Solution {

    public List<List<Integer>> combinationSum2(int[] candidates, int target) {

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

        Arrays.sort(candidates);

        helper(target, candidates, result, new ArrayList<Integer>(), 0);

        return result;

    }

    private void helper(int target, int[] candidates, List<List<Integer>> result, List<Integer> tmp, int index) {

        if(target == 0) {

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

            return;

        }

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

            return;

        }

        // Without local variable 'i', "Pick" branch must comes before "Not pick" branch

        // Pick

        tmp.add(candidates[index]);

        helper(target - candidates[index], candidates, result, tmp, index + 1);

        tmp.remove(tmp.size() - 1);

        // Not pick

        while(index + 1 < candidates.length && candidates[index] == candidates[index + 1]) {

            index++;

        }

        helper(target, candidates, result, tmp, index + 1);

    }

}

**Questions**:

**1. Why "target == 0 result.add, return" must come before "target < 0 || index >= candidates.length" ?**

**Wrong solution (based on Correction solution 2.3):**

class Solution {

    public List<List<Integer>> combinationSum2(int[] candidates, int target) {

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

        Arrays.sort(candidates);

        helper(target, candidates, result, new ArrayList<Integer>(), 0);

        return result;

    }

    private void helper(int target, int[] candidates, List<List<Integer>> result, List<Integer> tmp, int index) {

        // Return when "index >= candidates.length" first before "target == 0" when get one combination is wrong

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

            return;

        }

        if(target == 0) {

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

            return;

        }

        // Pick

        tmp.add(candidates[index]);

        helper(target - candidates[index], candidates, result, tmp, index + 1);

        tmp.remove(tmp.size() - 1);

        // Not pick

        while(index + 1 < candidates.length && candidates[index] == candidates[index + 1]) {

            index++;

        }

        helper(target, candidates, result, tmp, index + 1);

    }

}

Test out by:

Input: [2,5,2,1,2], 5

Output: [[1,2,2]]

Expected: [[1,2,2],[5]]

e.g.

candidates={2,5,2,1,2} -> sorted candidates={1,2,2,2,5},target=5(tg5),index=0(idx0)

/nt1                                                                                                                                                                                                \t1

                                                                                                            { }tg5,idx1                                                                                                                                                                                              {1}tg4,idx1

                                                  /nt2                                                                                                              \t2                                                                                                  /nt2                                                                        \t2

                                              { }tg5,idx2                                                                                                            {2}tg3,idx2                                                                                      {1}tg4,idx2                                                                      {1,2}tg2,idx2

                      /nt2                                                  \t2                                                    /nt2                                                          \t2                                          /nt2                                                    \t2                                      /nt2            \t2

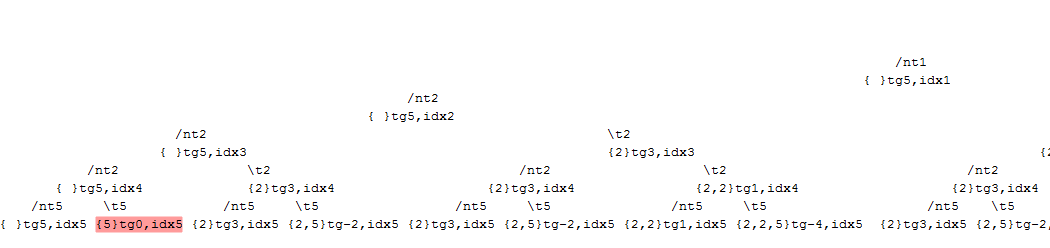
                    { }tg5,idx3                                            {2}tg3,idx3                                          {2}tg3,idx3                                                  {2,2}tg1,idx3                                {1}tg4,idx3                                            {1,2}tg2,idx3                        {1,2}tg2,idx3      {1,2,2}tg0,idx3

          /nt2                \t2                              /nt2                  \t2                              /nt2                    \t2                                /nt2                \t2                    /nt2                  \t2                                  /nt2                \t2                  /nt2      \t2

      { }tg5,idx4            {2}tg3,idx4                  {2}tg3,idx4              {2,2}tg1,idx4                  {2}tg3,idx4                {2,2}tg1,idx4                    {2,2}tg1,idx4        {2,2,2}tg-1,idx4      {1}tg4,idx4              {1,2}tg2,idx4                    {1,2}tg2,idx4    {1,2,2}tg0,idx4    {1,2}tg2,idx4 {1,2,2}tg0,idx4

    /nt5    \t5            /nt5    \t5                /nt5    \t5              /nt5    \t5                    /nt5    \t5                /nt5      \t5                /nt5        \t5                              /nt5    \t5            /nt5        \t5                /nt5        \t5                        /nt5      \t5

{ }tg5,idx5{5}tg0,idx5{2}tg3,idx5{2,5}tg-2,idx5 {2}tg3,idx5 {2,5}tg-2,idx5 {2,2}tg1,idx5 {2,2,5}tg-4,idx5  {2}tg3,idx5 {2,5}tg-2,idx5  {2,2}tg1,idx5 {2,2,5}tg-4,idx5  {2,2}tg1,idx5{2,2,5}tg-4,idx5            {1}tg4,idx5 {1,5}tg-1,idx5{1,2}tg2,idx5 {1,2,5}tg-3,idx5  {1,2}tg2,idx5 {1,2,5}tg-3,idx5      {1,2}tg2,idx5 {1,2,5}tg-3,idx5



**{5}tg0,idx5 missing explain all,** when index == candidates.length = 5, if we directly return based on "index >= candidates.length" like below

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

            return;

        }

        if(target == 0) {

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

            return;

        }

We will miss to add solution {5} into result for target == 0, one solution is switching the order of above two base cases which means when target == 0 we find a solution and avoid check on index against candidates.length.

The hard part is we cannot remove check on index against candidates.length, it will cause index out of boundary issue. The similar problem L90.Subsets II doesn't have this trouble, since it not check target, when index >= nums.length, means the one combination found.

**2. What will happen if not skip duplicate elements Backtracking style 2 recursion step by step (to view stretch on Notepad++) ?**

candidates={1,2,2,2,5},target=5(tg5),index=0(idx0)

branch ending condition: (1)index >=5 -> idx5 OR (2)target < 0 -> tg(<0)

combination found: target == 0(tg0)

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

What will happen if not skip duplicate elements Backtracking style 2 recursion step by step (to view stretch on Notepad++) ?

(1) tg0 -> one {5} on not take 1 branch, three {1,2,2} on take 1 branch

(2) Both not take and take certain element branch needs skip duplicates as we can see duplicate subtrees happen on both not take and take branch, so no need to create local variable 'i' to isolate skip duplicates process only on not take element branch like L90 Backtracking style 2

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

                                                                                                                                                                                                                { }tg5,idx0

                                                                                                                /nt1                                                                                                                                                                                                \t1

                                                                                                            { }tg5,idx1                                                                                                                                                                                              {1}tg4,idx1

                                                  /nt2                                                                                                              \t2                                                                                                  /nt2                                                                        \t2

                                              { }tg5,idx2                                                                                                            {2}tg3,idx2                                                                                      {1}tg4,idx2                                                                      {1,2}tg2,idx2

                      /nt2                                                  \t2                                                    /nt2                                                          \t2                                          /nt2                                                    \t2                                      /nt2            \t2

                    { }tg5,idx3                                            {2}tg3,idx3                                          {2}tg3,idx3                                                  {2,2}tg1,idx3                                {1}tg4,idx3                                            {1,2}tg2,idx3                        {1,2}tg2,idx3      {1,2,2}tg0,idx3

          /nt2                \t2                              /nt2                  \t2                              /nt2                    \t2                                /nt2                \t2                    /nt2                  \t2                                  /nt2                \t2                  /nt2      \t2

      { }tg5,idx4            {2}tg3,idx4                  {2}tg3,idx4              {2,2}tg1,idx4                  {2}tg3,idx4                {2,2}tg1,idx4                    {2,2}tg1,idx4        {2,2,2}tg-1,idx4      {1}tg4,idx4              {1,2}tg2,idx4                    {1,2}tg2,idx4    {1,2,2}tg0,idx4    {1,2}tg2,idx4 {1,2,2}tg0,idx4

    /nt5    \t5            /nt5    \t5                /nt5    \t5              /nt5    \t5                    /nt5    \t5                /nt5      \t5                /nt5        \t5                              /nt5    \t5            /nt5        \t5                /nt5        \t5                        /nt5      \t5

{ }tg5,idx5{5}tg0,idx5{2}tg3,idx5{2,5}tg-2,idx5 {2}tg3,idx5 {2,5}tg-2,idx5 {2,2}tg1,idx5 {2,2,5}tg-4,idx5  {2}tg3,idx5 {2,5}tg-2,idx5  {2,2}tg1,idx5 {2,2,5}tg-4,idx5  {2,2}tg1,idx5{2,2,5}tg-4,idx5            {1}tg4,idx5 {1,5}tg-1,idx5{1,2}tg2,idx5 {1,2,5}tg-3,idx5  {1,2}tg2,idx5 {1,2,5}tg-3,idx5      {1,2}tg2,idx5 {1,2,5}tg-3,idx5

**Refer to Deepseek**

**There is a similar problem as Leetcode 40. Combination Sum II, what's the major difference between Leetcode 39 and 40 ?**

The major difference between Leetcode 39 (Combination Sum) and Leetcode 40 (Combination Sum II) lies in the constraints and rules for forming combinations. Here's a detailed comparison:

**Leetcode 39 (Combination Sum):**

Input:

candidates: An array of distinct or non-distinct integers.

target: The target sum to achieve.

Rules:

Each number in candidates can be used unlimited times in the combinations.

The same combination (e.g., [2, 2, 3]) is allowed as long as it sums to the target.

Example:

Input: candidates = [2, 3, 6, 7], target = 7

Output: [[2, 2, 3], [7]]

Explanation:

2 can be used multiple times to form [2, 2, 3].

7 is used once to form [7].

Backtracking Approach:

In the recursive function, the same candidate can be reused by passing the same index i to the next recursion (e.g., helper(target - candidates[i], ..., i)).

**Leetcode 40 (Combination Sum II):**

Input:

candidates: An array of integers that may contain duplicates.

target: The target sum to achieve.

Rules:

Each number in candidates can be used only once in the combinations.

The solution set must \*\*not contain duplicate combinations\*\* (e.g., [1, 2, 2] and [2, 1, 2] are considered the same and should not both appear in the result).

Example:

Input: candidates = [10, 1, 2, 7, 6, 1, 5], target = 8

Output: [[1, 1, 6], [1, 2, 5], [1, 7], [2, 6]]

Explanation:

Each number is used only once.

Duplicate combinations like [1, 2, 5] and [2, 1, 5] are avoided.

Backtracking Approach:

To avoid reusing the same candidate, the next recursion starts from i + 1 (e.g., helper(target - candidates[i], ..., i + 1)).

To avoid duplicate combinations, skip over duplicate candidates in the sorted array (e.g., if candidates[i] == candidates[i - 1], skip candidates[i]).

**Key Differences:**

|  |  |  |
| --- | --- | --- |
| Feature | Leetcode 39 (Combination Sum) | Leetcode 40 (Combination Sum II) |
| **Reuse of Candidates** | Yes (unlimited times) | No (each candidate used only once) |
| **Duplicate Combinations** | Allowed (e.g., [2, 2, 3]) | Not allowed (e.g., [1, 2, 5] and  [2, 1, 5] are duplicates) |
| **Input Constraints** | May contain duplicates | May contain duplicates |
| **Backtracking Logic** | Reuse same index (i) | Move to next index (i + 1) |
| **Sorting Requirement** | Not required | Required to skip duplicates |

**Backtracking Code for Leetcode 40:**

Here’s how the backtracking solution for Leetcode 40 differs from Leetcode 39:

class Solution {

public List<List<Integer>> combinationSum2(int[] candidates, int target) {

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

Arrays.sort(candidates); // Sort to handle duplicates

helper(target, candidates, result, new ArrayList<>(), 0);

return result;

}

private void helper(int target, int[] candidates, List<List<Integer>> result, List<Integer> tmp, int index) {

if (target == 0) {

result.add(new ArrayList<>(tmp)); // Add valid combination

return;

}

if (target < 0) {

return; // Invalid path

}

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

// Skip duplicates to avoid duplicate combinations

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

continue;

}

tmp.add(candidates[i]);

// Move to next index to avoid reusing the same candidate

helper(target - candidates[i], candidates, result, tmp, i + 1);

tmp.remove(tmp.size() - 1); // Backtrack

}

}

}

**Why Sorting and Skipping Duplicates in Leetcode 40:**

Sorting ensures that duplicate candidates are adjacent, making it easy to skip them.

Skipping duplicates ensures that the same combination is not added multiple times (e.g., [1, 2, 5] and [2, 1, 5] are treated as the same).

**Example Walkthrough for Leetcode 40:**

Input: candidates = [10, 1, 2, 7, 6, 1, 5], target = 8

Sort the array: [1, 1, 2, 5, 6, 7, 10].

Explore combinations:

Start with 1 → tmp = [1], target 7.

Recurse with 1 → tmp = [1, 1], target 6.

Recurse with 2 → tmp = [1, 1, 2], target 4 (invalid).

Recurse with 5 → tmp = [1, 1, 5], target 1 (invalid).

Recurse with 6 → tmp = [1, 1, 6], target 0 ✅ Add [1, 1, 6].

Skip the second 1 (duplicate).

Start with 2 → tmp = [2], target 6.

Recurse with 5 → tmp = [2, 5], target 1 (invalid).

Recurse with 6 → tmp = [2, 6], target 0 ✅ Add [2, 6].

Start with 5 → tmp = [5], target 3 (invalid).

Start with 7 → tmp = [7], target 1 (invalid).

Result: [[1, 1, 6], [1, 2, 5], [1, 7], [2, 6]].

**Conclusion:**

Leetcode 39: Allows reuse of candidates and does not require sorting or duplicate handling.

Leetcode 40: Does not allow reuse of candidates, requires sorting, and skips duplicates to avoid duplicate combinations.

**Please elaborate how below if condition helps to avoid duplicate combinations ?**

// Skip duplicates to avoid duplicate combinations

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

continue;

}

Okay, let's try to figure out why that if condition is there in the code for Leetcode 40. So, the problem is Combination Sum II, and the key difference from Combination Sum I is that each candidate can be used only once, and the solution must not have duplicate combinations.

The code sorts the candidates first. That makes sense because sorting helps group duplicates together. But why check if i > index and candidates[i] == candidates[i-1]?

Let me think of an example. Suppose the sorted candidates are [1, 1, 2, 5], and the target is 8. If we don't have that condition, what happens? Let's say in the first step, index is 0. The loop starts at i=0 (the first 1). We add it to the temp list, then recursively call with index i+1=1. Now, in the next call, the loop starts at i=1 (second 1). Since we can't use the same candidate again (each can be used once), **but wait, no—each element in the array is considered a unique candidate, even if they have the same value. So using the second 1 is allowed as long as it's a different element.** But the problem states that the solution should not contain duplicate combinations. So if we have two 1s, choosing the first 1 and then the second 1 in different steps might lead to the same combination.

Wait, but in the code for Combination Sum II, after sorting, the duplicates are adjacent. So if we have [1,1,2], and target 3, possible combinations are [1,2] (using the first 1 and 2) and [1,2] (using the second 1 and 2). But these are duplicates and should be considered the same. So the code needs to avoid generating both.

The if condition says: if i is greater than the current index (meaning it's not the first element in this recursive call) and the current candidate is equal to the previous one, skip it. So in the first level of recursion (index=0), i starts at 0. The first 1 is picked. Then, when i=1, since candidates[1] == candidates[0], and i > index (0), we skip. So the second 1 is skipped. That way, even though there are two 1s, we only use the first one in this position, preventing duplicate combinations.

**Wait, but how does that work in different levels of recursion? Let's take the example where after picking the first 1, the next call has index=1. Then, in that call, i starts at 1. The previous element is candidates[0], but since i == index (1), the condition i > index is false, so we don't skip. So the second 1 can be included in a combination where the first 1 was picked. But that's allowed because the two 1s are different elements. However, if the code didn't skip when i > index, then in cases where there are multiple duplicates, the same value could be picked again in the same position (same recursion level), leading to duplicate combinations.**

Hmm, maybe I should walk through an example step by step. Let's take candidates = [1,1,2], target = 3. After sorting, it's [1,1,2].

The code starts with index=0. Loop over i from 0 to 2.

At i=0, add 1 to temp. Target becomes 2. Then call helper with index=1 (since next recursion starts at i+1=1).

In the helper for target=2 and index=1:

Loop i from 1 to 2.

At i=1, check if i > index (1>1? No). So proceed. Add 1. Target becomes 1. Call helper with index=2.

In helper for target=1 and index=2:

Loop i from 2. candidates[2] is 2. Add 2. Target becomes -1. Backtrack. Then, try next candidate. So, no valid path here. Backtrack.

Back to i=1 in the previous level. Remove 1. Now i increments to 2. Add 2. Target becomes 0. So the combination [1,2] is added.

Now, back to the initial call, after processing i=0, we remove 1 and move to i=1.

At i=1 in the initial loop (index=0), check if i>0 and candidates[1] == candidates[0]. Yes. So skip this i. Thus, the second 1 is skipped at the first level. So the code doesn't process the second 1 as the starting element. This prevents having two combinations starting with 1, leading to duplicates.

**So, by skipping duplicates when i is greater than the starting index, the code ensures that in the same level of recursion (same position in the combination), we don't pick the same value multiple times. But when we are in a deeper recursion (different position), duplicates can be picked as long as they are different elements but not in the same position.**

Therefore, the condition i > index ensures that we skip duplicates only at the same level, allowing duplicates in different positions but not reusing the same value in the same position, which would create duplicate combinations.

So the key idea is that in the same recursive level (same position in the combination building), we skip over duplicate candidates to avoid creating duplicate combinations. Because if we have multiple same candidates in the same position, they would generate the same combination structure. By skipping them, we ensure each unique combination is generated once.

The if condition in the code for Leetcode 40 (Combination Sum II) is critical for avoiding duplicate combinations. Here’s a detailed explanation of how it works:

**Key Purpose of the if Condition:**

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

continue;

}

This condition ensures that duplicate combinations are skipped when candidates have the same value. It leverages sorting and checks for duplicates at the same recursion level to prevent redundant exploration.

**Why It Works:**

Sorting First:

The candidates are sorted (e.g., [1, 1, 2, 5] instead of [1, 2, 1, 5]). This groups duplicates together, making it easier to detect and skip them.

Skipping Duplicates at the Same Level:

The condition i > index ensures we only skip duplicates \*\*after the first occurrence\*\* in the current recursive level.

For example, in a sorted array [1, 1, 2, 5], when index = 0 (first level of recursion), i starts at 0:

At i = 0, we pick the first 1 and proceed to explore combinations starting with 1.

At i = 1, since candidates[1] == candidates[0] and i > index (1 > 0), we skip this 1. This avoids starting another combination with the same value at the same level, preventing duplicates like [1, 2, 5] and [1, 2, 5] (from the second 1).

Allowing Duplicates in Deeper Recursion:

Once we move to deeper recursion levels (e.g., after picking the first 1 at index = 0), the next recursion starts at index + 1 = 1. At this new level, i starts at 1, and candidates[1] (the second 1) is allowed because i == index (1 == 1). This lets us form combinations like [1, 1, 2] (valid if the sum matches).

**Example Walkthrough:**

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

Sorted Candidates: [1, 1, 2]

Valid Combinations: [1, 2] (using either 1) and [1, 1, 1] (invalid for target 3).

First Level (index = 0):

i = 0: Pick 1 → tmp = [1], target becomes 2.

Recurse with index = 1 (next level).

i = 1: Check i > 0 and candidates[1] == candidates[0] → skip.

Second Level (index = 1):

i = 1: Pick the second 1 → tmp = [1, 1], target becomes 1.

Recurse with index = 2 (next level).

i = 2: Pick 2 → tmp = [1, 1, 2], target becomes -1 → backtrack.

Third Level (index = 2):

i = 2: Pick 2 → tmp = [1, 2], target becomes 0 → valid combination added.

Result: [[1, 2]] (no duplicate [1, 2] from the second 1).

**Why This Condition is Necessary:**

Without this check:

For candidates = [1, 1, 2], you’d generate two identical combinations [1, 2] (one from each 1).

The condition ensures that only the first occurrence of a duplicate value is used at each recursion level, preventing redundant exploration.

**Summary:**

Sorting: Groups duplicates for easy detection.

i > index: Ensures we only skip duplicates after the first occurrence in the current recursion level.

candidates[i] == candidates[i - 1]: Detects duplicates in the sorted array.

Result: Unique combinations are generated, and duplicates are avoided.

**What's the difference between Leetcode 40 and Leetcode 377 ?**

Okay, let's see. The user is asking about the differences between LeetCode 40 (Combination Sum II) and LeetCode 377 (Combination Sum IV). I remember that both problems are about finding combinations that sum up to a target, but there must be some key differences. Let me think step by step.

**First, LeetCode 40: Combination Sum II. The problem allows each candidate to be used only once, and the solution must not contain duplicate combinations. Also, the input array can have duplicates. So, for example, if the input is [10,1,2,7,6,1,5] and target 8, the output includes [1,1,6], [1,2,5], etc., but not [1,2,5] and [2,1,5] as separate entries. The approach here involves sorting the array to handle duplicates and skipping over the same elements in the recursion to avoid duplicates in the result.**

**Then LeetCode 377: Combination Sum IV. This one is different because it's about permutations. The problem counts different sequences as different combinations if the order is different. For example, [1,1,2] and [1,2,1] are considered distinct. Also, the candidates can be reused, but since order matters, it's more about permutations that sum to the target. The solution here is often approached with dynamic programming, where you track the number of ways to reach each possible sum up to the target.**

Wait, but in LeetCode 40, each number can be used only once, and in 377, numbers can be reused, right? Or is that different? Let me check. No, wait. LeetCode 377's problem statement allows using numbers multiple times. Wait, no. Wait, no. Let me confirm. Combination Sum IV (377) allows using the same number multiple times, just like Combination Sum (LeetCode 39), but the key difference is that order matters. So the count is the number of possible permutations that sum to the target. For example, with candidates [1,2,3] and target 4, the answer includes 1+1+1+1, 1+1+2 (in any order), etc., and each different order is considered a different combination.

But in LeetCode 40, each candidate can be used only once, and the problem is about combinations (order doesn't matter), and the array can have duplicates. So the solution must avoid both reusing the same element and avoid duplicate combinations in the output.

So the main differences would be:

1. Element Reuse: LeetCode 40 does not allow reuse (each element once), LeetCode 377 allows reuse (unlimited times).

2. Order Matters: LeetCode 40 considers combinations (order doesn't matter), so [1,2] and [2,1] are the same. LeetCode 377 counts permutations (order matters), so they are different.

3. Handling Duplicates: LeetCode 40's input can have duplicates, so the solution must sort and skip duplicates. LeetCode 377's input is unique, but since order matters, different permutations are allowed even if they use the same elements in different orders.

4. Solution Approach: LeetCode 40 uses backtracking with sorting and skipping duplicates. LeetCode 377 can use dynamic programming to count permutations, as each step can choose any candidate and build up the sum.

Wait, but the user might be mixing up the problem constraints. Let me check the actual problem statements again.

LeetCode 40: Combination Sum II

- Candidates may contain duplicates.

- Each number in candidates may only be used once in the combination.

- The solution set must not contain duplicate combinations.

- Target is a positive integer.

LeetCode 377: Combination Sum IV

- The array consists of distinct positive integers.

- Combinations are considered different if the order of numbers differs.

- The same number can be chosen multiple times.

- The problem asks for the number of possible combinations (permutations) that add up to the target.

So the major differences are:

- Reuse of Elements: 40 does not allow reuse (each used once), 377 allows reuse.

- Order Significance: 40's combinations are order-agnostic (sets), 377's are sequences (order matters).

- Input Constraints: 40's candidates can have duplicates, 377's are distinct.

- Output Requirements: 40 avoids duplicate combinations, 377 counts all permutations.

**The approaches differ accordingly. For 40, you sort the array, backtrack, and skip duplicates. For 377, you use DP where dp[i] is the number of ways to reach sum i by adding any candidate to previous sums, allowing permutations.**

Another point is that 377 can have a very large target, making a pure backtracking solution infeasible due to time constraints, hence the need for DP with memoization or bottom-up approach.

So summarizing, the key differences are reuse, order, duplicates in input, and the approach used (backtracking vs DP).

The key differences between LeetCode 40 (Combination Sum II) and LeetCode 377 (Combination Sum IV) lie in their constraints, problem definitions, and solution approaches. Here's a structured breakdown:

**1. Problem Definition & Constraints**

|  |  |  |
| --- | --- | --- |
| Feature | LeetCode 40 (Combination Sum II) | LeetCode 377 (Combination Sum IV) |
| **Element Reuse** | Each candidate can be used once per combination. | Candidates can be reused unlimited times. |
| **Order Significance** | Combinations are order-agnostic (e.g., [1,2] and [2,1] are duplicates). | Permutations are order-sensitive (e.g., [1,2] and [2,1] are distinct). |
| **Input Constraints** | Candidates may contain duplicates. | Candidates are distinct integers. |
| **Output Requirements** | Must avoid duplicate combinations. | Must count all permutations. |

**2. Example Scenarios**

**LeetCode 40:**

Input: candidates = [10,1,2,7,6,1,5], target = 8

Valid Combinations: [[1,1,6], [1,2,5], [1,7], [2,6]]

Explanation:

[1,2,5] is allowed, but [2,1,5] is considered a duplicate and excluded.

Each candidate is used once per combination.

**LeetCode 377:**

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

Valid Permutations: [1,1,1,1], [1,1,2], [1,2,1], [2,1,1], [2,2], [1,3], [3,1], ...

Explanation:

Different orders of the same elements (e.g., [1,1,2] and [1,2,1]) are counted separately.

**3. Solution Approaches**

**LeetCode 40 (Combination Sum II):**

Backtracking with Sorting & Skipping Duplicates:

Sort the input to group duplicates.

Use backtracking to explore combinations.

Skip duplicates in the same recursion level (e.g., if candidates[i] == candidates[i-1] and i > current\_index, skip candidates[i]).

Ensure each candidate is used once per combination.

**LeetCode 377 (Combination Sum IV):**

Dynamic Programming (DP):

Define dp[target + 1] where dp[i] = number of permutations summing to i.

Initialize dp[0] = 1 (one way to reach sum 0: use no elements).

For each sum i from 1 to target, iterate through candidates:

If i >= candidate, update dp[i] += dp[i - candidate].

Permutations are counted by allowing any candidate to extend any previous sum.

**5. When to Use Each Approach**

LeetCode 40: Use backtracking with sorting and duplicate skipping for problems where order doesn’t matter, elements are unique per combination, and input has duplicates.

LeetCode 377: Use dynamic programming for problems where order matters and elements can be reused (permutations).

**Summary**

|  |  |  |
| --- | --- | --- |
| Aspect | LeetCode 40 (Combination Sum II) | LeetCode 377 (Combination Sum IV) |
| **Reuse of Elements** | No | Yes |
| **Order Sensitivity** | No (combinations) | Yes (permutations) |
| **Input Duplicates** | Yes | No |
| **Solution Approach** | Backtracking + Sorting + Skip Duplicates | Dynamic Programming |
| **Time Complexity** | Exponential (with pruning) | O(target × n) (efficient with DP) |

Choose LeetCode 40 for unique combinations with no reuse, and LeetCode 377 for counting permutations with reuse.

**Refer to**

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

[L377.P16.4.Combination Sum IV(Backpack VI) (Ref.L39,L40,L322)](note://8000F8241A0B4F3AA39E5CA16B6A7BB8)