<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) {

return;

}

if(target == 0) {

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

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.

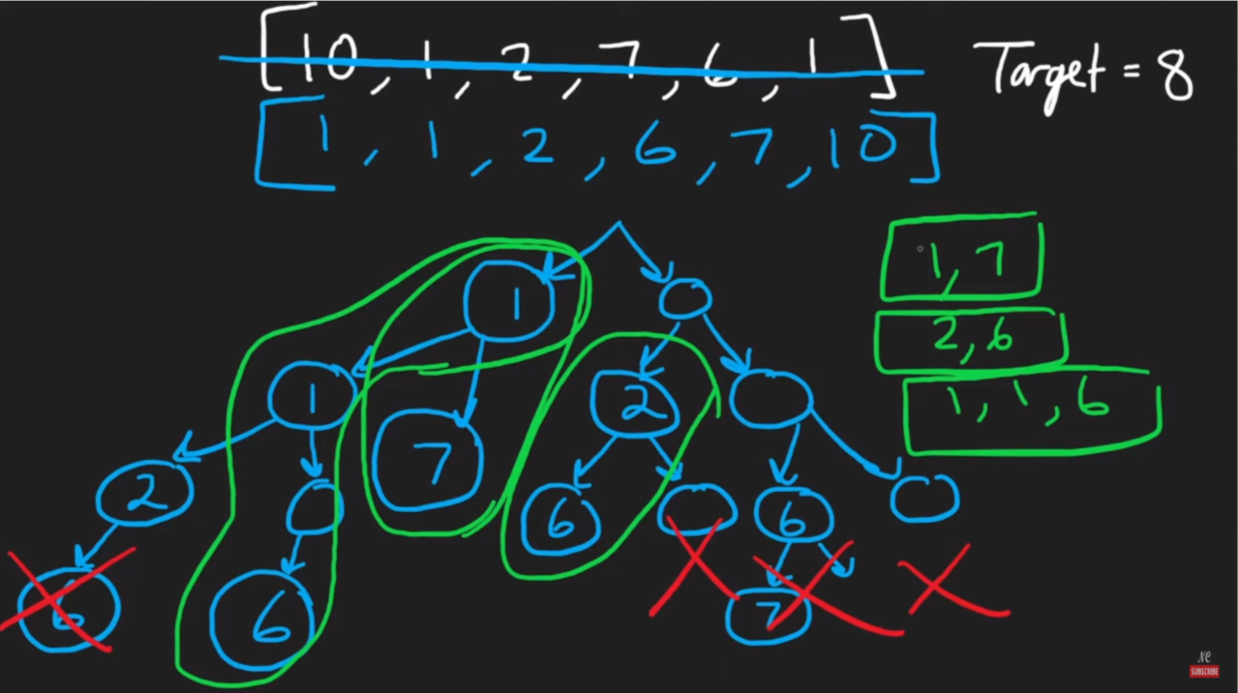
Space Complexity: O(length\_of\_longest\_combination)

**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:**

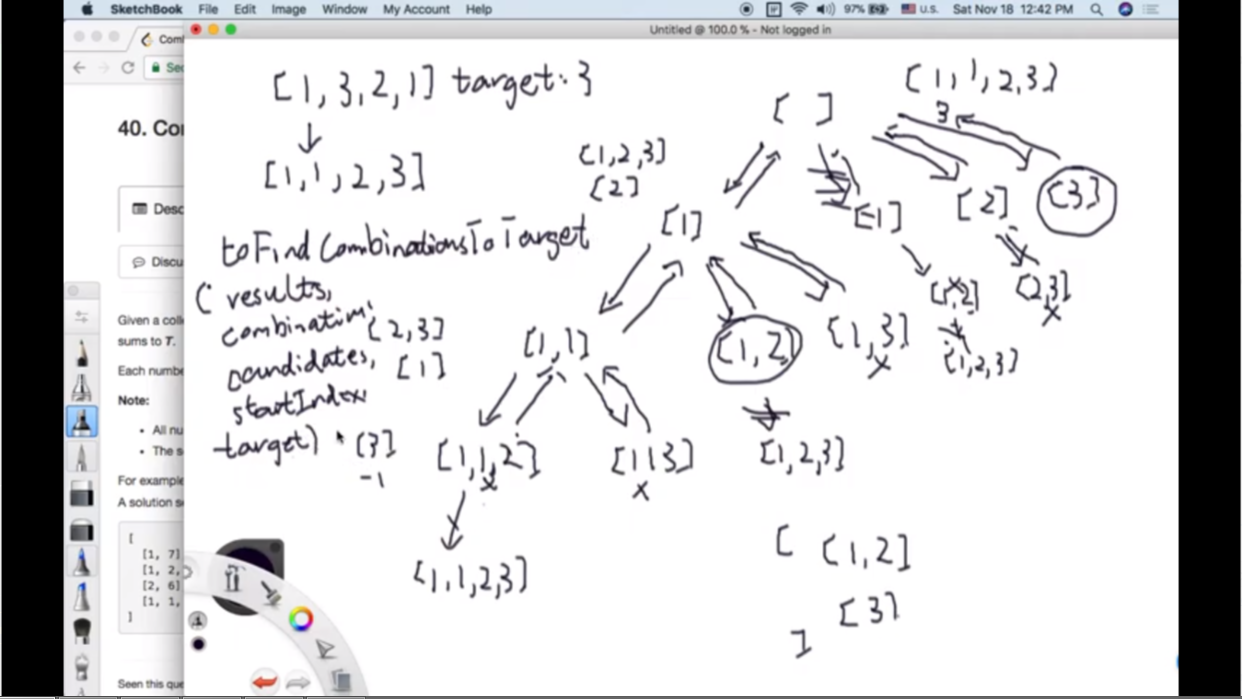


**Another video explain**

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

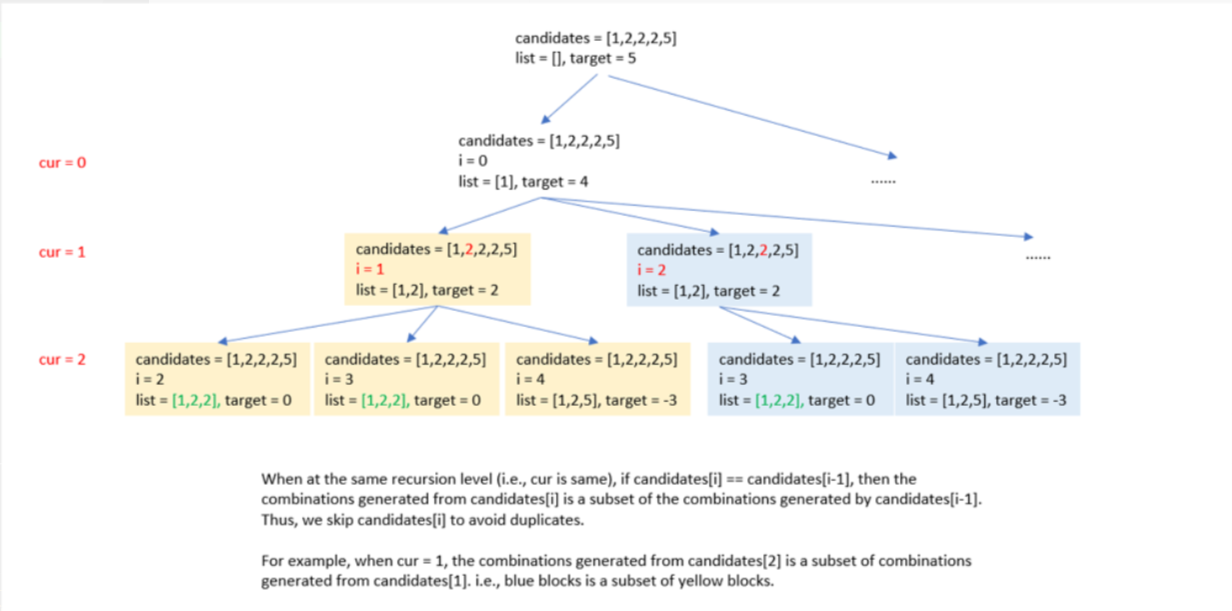
<https://www.youtube.com/watch?v=j9_qWJClp64>

**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);

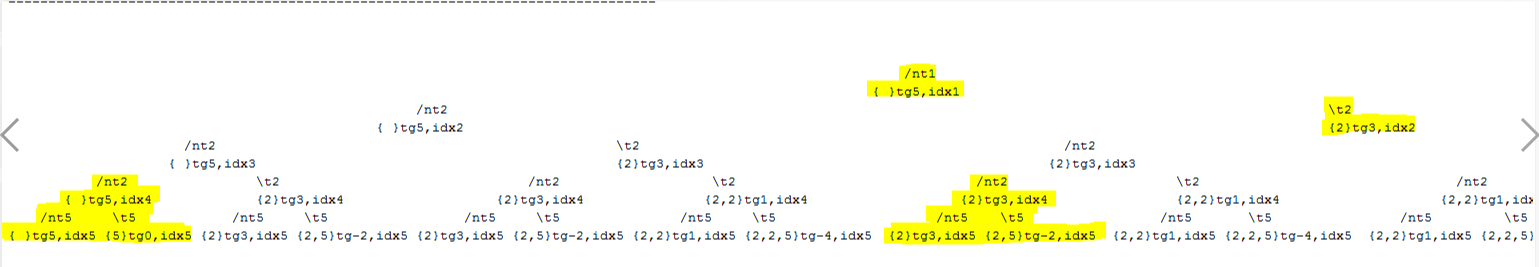
tmp.remove(tmp.size() - 1);

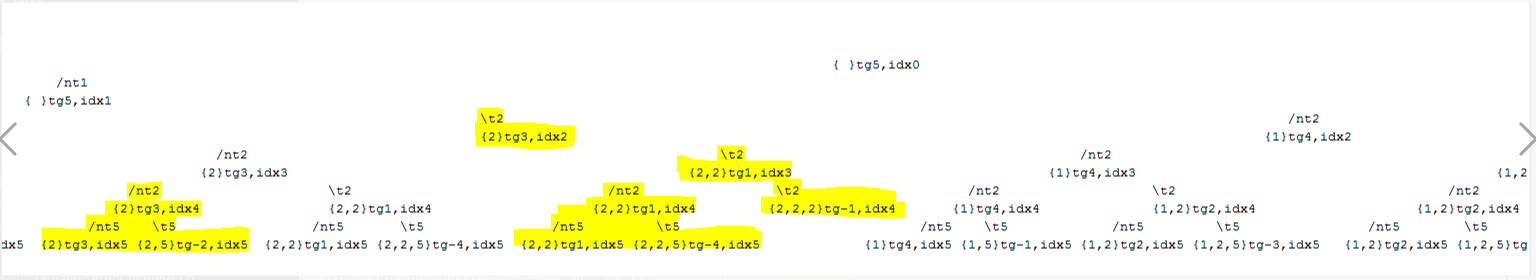
}

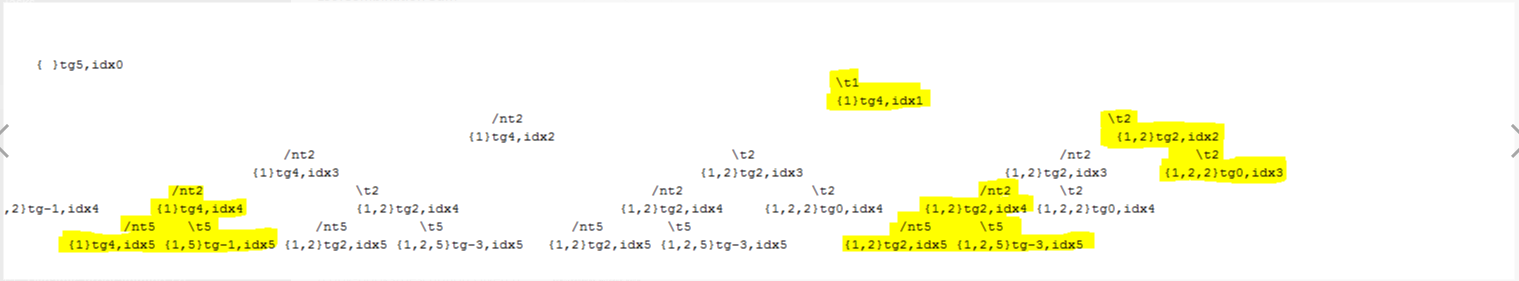
}

**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;

}

void backtrack(vector<int>& candidates, int target, vector<vector<int>>& ans, vector<int>& curr, int index) {

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