

■ Combination Sum II

Each element used at most once — no duplicate combinations

● Recursion

■ MEDIUM

■ What You Will Learn

- The difference from Combination Sum I: no element reuse
- Combining sort + skip-duplicate logic from Subsets II
- Why $i+1$ prevents reuse and how to handle duplicates
- Two tricks working together: no-reuse + no-duplicate

■ Intuition First

candidates=[10,1,2,7,6,1,5], target=8. Elements cannot be reused. But there are two 1s — you can use each 1 once, but you shouldn't produce the same combination from both 1s. Sort first ([1,1,2,5,6,7,10]), use $i+1$ to prevent reuse, and skip duplicate at same level to prevent duplicate combos.

■ Two Rules Together

Rule 1: Pass $i+1$ (not i) → each element used at most once. Rule 2: if ($i > \text{start}$ && $\text{cand}[i] == \text{cand}[i-1]$) → skip duplicate at same level. Both rules together give unique combinations with no reuse.

■ Visual — candidates=[1,1,2,5,6,7,10], target=8

Sort: [1,1,2,5,6,7,10]

```
try index 0 (val=1): curr=[1], rem=7
  try index 1 (val=1): curr=[1,1], rem=6
    try index 2 (val=2): curr=[1,1,2], rem=4
      try index 3 (val=5): 5>4 → PRUNE (sorted!)
    try index 3 (val=5): curr=[1,1,5], rem=1
      try index 4 (val=6): 6>1 → PRUNE
    try index 4 (val=6): curr=[1,1,6]=8 → ADD ✓
  try index 2 (val=2): i=2>start=1 and cand[2]=2!=cand[1]=1 → OK
    ... eventually finds [1,2,5]=8 ✓
  try index 3 (val=5): curr=[1,5], rem=2
    ... [1,5,2]? No, 2 < 5 (sorted), won't appear. Skip.
  try index 5 (val=7): curr=[1,7]=8 → ADD ✓
try index 1 (val=1): i=1>start=0 and cand[1]==cand[0] → SKIP!
try index 2 (val=2): curr=[2], rem=6
  try index 3 (val=5): curr=[2,6]→ no wait [2,5] rem=1→PRUNE
  try index 4 (val=6): curr=[2,6]=8 → ADD ✓
```

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

■ Java Code

```
1  class Solution {
2      public List<List<Integer>> combinationSum2(int[] candidates, int target) {
3          Arrays.sort(candidates); // CRITICAL: sort for duplicate detection
4          List<List<Integer>> result = new ArrayList<>();
5          backtrack(candidates, target, 0, new ArrayList<>(), result);
6          return result;
7      }
8
9      private void backtrack(int[] cand, int remaining, int start,
10         List<Integer> curr, List<List<Integer>> result) {
11          if (remaining == 0) {
12              result.add(new ArrayList<>(curr));
13              return;
14          }
15
16          for (int i = start; i < cand.length; i++) {
17              if (cand[i] > remaining) break; // Pruning
18
19              // Skip duplicate elements at SAME recursion level
20              if (i > start && cand[i] == cand[i-1]) continue;
21
22              curr.add(cand[i]);
23              backtrack(cand, remaining - cand[i], i+1, curr, result); // i+1!
24              curr.remove(curr.size() - 1);
25          }
26      }
27  }
```

■■ Time & Space Complexity

Type	Value	What It Means
Time	$O(2^n)$	Worst case: all unique elements $\rightarrow 2^n$ subsets
Space	$O(n)$	Recursion depth and temp list = $O(n)$

■■ Common Mistakes

■■ i vs i+1

Passing `i` allows reuse (Combo Sum I). Passing `i+1` prevents reuse (Combo Sum II). One character difference, huge behavioral difference.

■■ Skip Condition

`if(i>0 && ...)` vs `if(i>start && ...)`. Use `i>start` to only skip at the same recursion level, not globally.

■ Practice Problems

#	Problem	Diff
1	Combination Sum II (LeetCode 40)	Medium
2	Combination Sum (LeetCode 39)	Medium
3	Subsets II (LeetCode 90)	Medium
4	3Sum (LeetCode 15)	Medium

■ Quick Revision

✓	Sort first to group duplicates adjacent
✓	i+1 in recursive call → no element reuse
✓	if(i>start && cand[i]==cand[i-1]) continue → no duplicate combos
✓	Prune: if cand[i] > remaining, break
✓	Combo I vs II: only difference is i vs i+1 in recursive call