

■ Combination Sum I

Find all combinations that add up to target — elements can repeat

● Recursion

■ MEDIUM

■ What You Will Learn

- Backtracking with a running sum toward a target
- Allowing repeated use of the same element
- Pruning the search tree when sum exceeds target
- How to avoid duplicate combinations in output

■ Intuition First

candidates=[2,3,6,7], target=7. You want all combos (with repetition allowed) that sum to 7. Think of it as filling a jar: try adding 2, keep adding 2s until you overflow, backtrack, try 3, etc. You're exploring a tree of possibilities and cutting branches that go over the target.

■ Key Difference from Subsets

In Combination Sum I, the SAME element can be used multiple times. So when we recurse, we pass 'i' (same index) instead of 'i+1'. We only move forward (never backward) to avoid duplicate combinations.

■ Visual — candidates=[2,3,6,7], target=7

```
start=0, sum=0, curr=[]
try 2: sum=2, curr=[2]
  try 2: sum=4, curr=[2,2]
    try 2: sum=6, curr=[2,2,2]
      try 2: sum=8 > 7 → PRUNE (stop this branch)
      try 3: sum=9 > 7 → PRUNE
      try 3: sum=7 = 7 → ADD [2,2,3] ✓
      try 6: sum=12 > 7 → PRUNE
    try 3: sum=7 = 7 → ADD [2,5]... wait
    try 3: sum=5+3=8 > 7 → PRUNE
  try 6: sum=8 > 7 → PRUNE
try 3: sum=3, curr=[3]
  try 3: sum=6 → try 3: sum=9>7 PRUNE. try others...
  try 4: ... try 7: sum=3+7=10>7 PRUNE
try 6: sum=6 → try 6:12>7 PRUNE. try 7:13>7 PRUNE
try 7: sum=7 = target → ADD [7] ✓
```

Result: [[2,2,3], [7]]

■ Java Code

```

1  class Solution {
2      public List<List<Integer>> combinationSum(int[] candidates, int target) {
3          List<List<Integer>> result = new ArrayList<>();
4          Arrays.sort(candidates); // Optional but helps pruning
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)); // Found a valid combo!
13             return;
14         }
15
16         for (int i = start; i < cand.length; i++) {
17             if (cand[i] > remaining) break; // Pruning: sorted, so skip rest
18
19             curr.add(cand[i]);
20             backtrack(cand, remaining - cand[i], i, curr, result); // i not i+1!
21             curr.remove(curr.size() - 1); // Backtrack
22         }
23     }
24 }

```

■ Time & Space Complexity

Type	Value	What It Means
Time	$O(2^{\text{target}})$	Each element can be used up to target/min(candidates) times
Space	$O(\text{target})$	Recursion depth up to target/min(element)

■ Common Mistakes

■ i+1 vs i

Passing i+1 means each element can only be used once. Passing i allows repeats. Combination Sum I uses i (repeats allowed).

■ Missing Pruning

Without 'if(cand[i]>remaining) break', the recursion explores many dead ends. Always prune when the current element already exceeds remaining target.

■ Practice Problems

#	Problem	Diff
1	Combination Sum (LeetCode 39)	Medium
2	Combination Sum II (LeetCode 40)	Medium
3	Combination Sum III (LeetCode 216)	Medium
4	Letter Combinations of a Phone Number (LeetCode 17)	Medium

■ Quick Revision

✓	Sort candidates for easy pruning
✓	Pass 'i' (not i+1) to allow reuse of same element
✓	Prune: if <code>cand[i] > remaining</code> , break loop
✓	Add snapshot when <code>remaining == 0</code>
✓	Backtrack: remove last added element after each recursive call