LeetCode 39 — Combination Sum

1. Problem Title & Link

- **Title:** LeetCode 39 — Combination Sum
- **Link:** https://leetcode.com/problems/combination-sum/

2. Problem Statement (Short Summary)

You are given:

- An array of **distinct** integers candidates
- A target integer target

Find **all unique combinations** of candidates where the chosen numbers **sum to target**.

You may use **each number unlimited times**.

Order of numbers inside a combination does NOT matter.

3. Examples (Input → Output)

Example 1

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

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

Example 2

Input: candidates = [2,3,5], target = 8

Output: [[2,2,2,2],[2,3,3],[3,5]]

Example 3

Input: candidates = [2], target = 1

Output: []

4. Constraints

- 1 <= candidates.length <= 30
- 1 <= candidates[i] <= 200
- All numbers are unique
- Unlimited usage of each number
- Must return unique combinations

5. Core Concept (Pattern / Topic)

Backtracking – Choose, Explore, Unchoose

- Explore combinations by picking candidates starting from a given index (so no duplicates).
- Allow re-use of same element by **not moving to next index** after picking.

6. Thought Process (Step-by-Step Explanation)

We use **DFS + backtracking**.

At any recursive call:

We track:

- current combination (path)
- remaining target
- start index (prevents permutations)

Steps:

1. If target == 0 → valid combination → add path to answer.
2. If target < 0 → invalid → stop exploring.
3. For each index from start to end:
   - Choose candidates[i]
   - Recurse with target - candidates[i]
   - After returning, remove last element (unchoose)

Why i (same index) instead of i+1?

Because numbers can be used unlimited times.

7. Visual / Intuition Diagram

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

```
Backtracking tree (major branches):
                          []
           /           |            \
          2            3              6           7
        /     \        |              |           |
     2         3       3              6           7
    / \        |       |              |
   2   3       3       3
  /     \      |
 2       3     3
 |         |
 3         3
 ↓         ↓
7 reached  7 reached
```

Valid:

[2,2,3]

[7]

8. Pseudocode

```
result = []
```

```
function backtrack(start, path, target):
    if target == 0:
        add copy(path) to result
        return
    if target < 0:
        return

    for i from start to len(candidates)-1:
        path.append(candidates[i])
        backtrack(i, path, target - candidates[i])   # reuse allowed
        path.pop()

call backtrack(0, [], target)
return result
```

9. Code Implementation

✅ **Python**

```
class Solution:
    def combinationSum(self, candidates: List[int], target: int) ->
List[List[int]]:
        res = []

        def backtrack(start, path, total):
            if total == target:
                res.append(path[:])
                return
            if total > target:
                return

            for i in range(start, len(candidates)):
                path.append(candidates[i])
                backtrack(i, path, total + candidates[i])
                path.pop()

        backtrack(0, [], 0)
        return res
```

✅ **Java**

```
class Solution {
```

```
    public List<List<Integer>> combinationSum(int[] candidates, int target) {
        List<List<Integer>> res = new ArrayList<>();
        backtrack(0, candidates, target, new ArrayList<>(), res);
        return res;
    }

    private void backtrack(int start, int[] candidates, int remaining,
                           List<Integer> path, List<List<Integer>> res) {
        if (remaining == 0) {
            res.add(new ArrayList<>(path));
            return;
        }
        if (remaining < 0) return;

        for (int i = start; i < candidates.length; i++) {
            path.add(candidates[i]);
            backtrack(i, candidates, remaining - candidates[i], path, res);
            path.remove(path.size() - 1);
        }
    }
}
```

10. Time & Space Complexity

Metric	Complexity
Time	O(2^t) worst case (tree branching)
Space	O(t) recursion stack + O(answer size)

11. Common Mistakes / Edge Cases

❌ Using i+1 instead of i → prevents reuse

❌ Sorting candidates (unnecessary but ok)

❌ Forgetting to pop the path after recursion

❌ Adding path directly instead of making a copy

Edge cases:

- target smaller than all candidates → return []
- one candidate equal to target

12. Detailed Dry Run (Step-by-Step)

Input:

candidates = [2,3,6,7]

target = 7

Start:

path=[], start=0, total=0

Try 2:

path=[2], total=2

Try 2 again:

path=[2,2], total=4

Try 2 again:

path=[2,2,2], total=6

Try 2 again:

path=[2,2,2,2], total=8 → >7 → backtrack

Try 3:

path=[2,2,3], total=7 → valid

Store → [2,2,3]

Backtrack to try next options.

Try 3 as first:

path=[3], total=3

path=[3,3], total=6

path=[3,3,3], total=9 → invalid

Try 6:

path=[6], total=6

path=[6,6], total=12 → invalid

Try 7:

path=[7], total=7 → valid

Final result:

[[2,2,3], [7]]

13. Common Use Cases

- Money/change formation
- Combination of items with repetition
- Unbounded knapsack background
- Recipe/mix formation problems

14. Common Traps

- Not passing correct start index
- Letting combinations get duplicated
- Forgetting to pop the last element
- Overcounting due to considering permutations

15. Builds To (Related Problems)

- **LC 40** — Combination Sum II (no repeats, duplicates allowed → harder)
- **LC 216** — Combination Sum III
- **LC 377** — Combination Sum IV (DP version)
- **LC 17** — Letter combinations of phone number (backtracking)
- **LC 78/90** — Subsets / Subsets with duplicates

16. Alternate Approaches + Comparison

Approach	Time	Space	Notes
Backtracking	exponential	O(target/depth)	✔ Best
DP (count ways)	O(n*target)	O(target)	Counts ways, not combinations
BFS	exponential	high	Rarely used

17. Why This Solution Works (Short Intuition)

The combination search tree explores adding each candidate unlimited times,
but by always moving forward from start → end, we avoid permutations
and generate only unique combinations.

18. Variations / Follow-Up Questions

- What if candidates have duplicates? (use LC 40 logic)
- What if numbers can only be used ONCE? (subset problem)
- What if order matters? (turn into DP counting problem)
- How to optimize with pruning? (sort + break early)