Topic Introduction

Today, we're diving into the world of **backtracking for generating combinations under constraints**. Backtracking is a powerful algorithmic technique used to solve problems that require searching through all possible configurations to find solutions that meet certain criteria.

What is Backtracking?
Backtracking is a depth-first search strategy for exploring possible choices, one step at a time. If a path leads to a solution, we keep going; if it leads to a dead end, we *backtrack* to the previous choice and try something else. Think of it as exploring a maze and marking your path, so you can retrace your steps if you hit a wall.

How Does It Work?
At its heart, backtracking involves:

- Building a solution incrementally.
- Abandoning a path ("backtracking") as soon as we determine it cannot possibly lead to a valid solution.
- Exploring all possible paths in a systematic way.

When and Why is it Useful?
Backtracking is ideal when:

- You need to generate all possible configurations (combinations, permutations, subsets).
- There are constraints that must be satisfied.
- Brute-force is possible but inefficient, and pruning paths early can save time.

Quick Example:
Suppose you want to generate all subsets of [1, 2, 3]. Using backtracking, you'd recursively decide for each number whether to include it or not, exploring all possible combinations. This approach is not only systematic but also allows you to easily add constraints (for example, only subsets with sum greater than 3).

Now, let's look at three classic problems that showcase backtracking for combinations, each with its own twist:

The Three Problems

- **Combination Sum (Leetcode 39):** Find all combinations of a set of numbers that sum up to a target. Each number can be used unlimited times.
- **Combination Sum II (Leetcode 40):** Like the first, but each number can be used at most once, and the list may have duplicates.
- **Combination Sum III (Leetcode 216):** Pick *k* numbers (from 1 to 9) that sum up to a target. Each number is used at most once.

Why are these grouped together?
All three are *combination* problems solved by backtracking, but each introduces slightly different constraints:

- The first allows unlimited reuse.
- The second handles duplicates and limits each number to one use.
- The third restricts the pool to 1-9 and fixes the number of picks.

We'll start with the most flexible (Combination Sum), then build up to the more constrained versions.

Problem 1: Combination Sum ([Leetcode 39](https://leetcode.com/problems/combination-sum/))

Problem Rephrased:

Given an array of positive integers (candidates) and a target number, find all unique combinations of candidates where the numbers sum to the target. Each number from candidates can be used *any number of times*. Return the combinations in any order.

Example Input/Output:

Input:

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

Output:

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

Explanation:

- [2, 2, 3] sums to 7 (using 2 three times and 3 once).
- [7] uses 7 directly.

Let's Think Through It:

Imagine you're trying to build a sum of 7. Start with zero, then try adding each candidate:

- Add 2: now at 2, can add 2 again (now at 4), again (6), but need one more, so try 3 (total 9, too high), backtrack.
- Alternatively, after [2, 2], try 3: [2, 2, 3] = 7.

Try to sketch out the process on paper if it feels confusing!

Additional Test Case to Try:

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

Expected output: [[2,2,2,2],[2,3,3],[3,5]]

Brute-Force Approach:

Try every possible combination of candidates, with replacement, to see if any sum to target. This would be extremely slow, especially for large targets, since the number of possible combinations grows rapidly.

- **Time Complexity:** Exponential in target and number of candidates.

Optimal Approach: Backtracking

- *Pattern*: Recursive backtracking, try each number starting from the current index (to allow unlimited reuse).
- *Logic*:
    - At each step, try adding each candidate (starting from current or later, to avoid duplicates like [2,3] and [3,2]).
    - If sum exceeds target, stop (prune).
    - If sum equals target, record combination.
    - Backtrack to try other possibilities.

Python Solution:

```python
def combinationSum(candidates, target):
    results = []

    def backtrack(remaining, start, path):
        # Base case: found a valid combination
        if remaining == 0:
            results.append(list(path))  # Make a copy of current path
            return
        # If the sum exceeds target, stop exploring this path
        if remaining < 0:
            return

        # Try each candidate from the current position
        for i in range(start, len(candidates)):
            # Include candidates[i] and continue
            path.append(candidates[i])
            # Because we can reuse the same number, we pass 'i' (not 'i + 1')
            backtrack(remaining - candidates[i], i, path)
            # Backtrack: remove last added number
            path.pop()

    backtrack(target, 0, [])
    return results
```

Time Complexity:

- Roughly O(2^target) in the worst case, since each recursive call can branch in multiple ways.
- **Space Complexity:** O(target) for recursion stack and result storage.

What Each Part Does:

- `results`: Stores all valid combinations.
- `backtrack(remaining, start, path)`: Recursive function.
  - `remaining`: Remaining value to reach target.
  - `start`: Index to start from (to avoid duplicates).
  - `path`: Current combination being built.
- For each candidate, add it, recurse, then remove it (backtrack).

Trace Example:

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

- Start with []
- Add 2: [2], remaining: 5
  - Add 2: [2,2], remaining: 3
    - Add 2: [2,2,2], remaining: 1
      - Add 2: [2,2,2,2], remaining: -1 (too high, backtrack)
      - Add 3: [2,2,2,3], remaining: -2 (too high, backtrack)

        - Add 6: ...
        - Add 7: ...
    - Add 3: [2,2,3], remaining: 0 (success! Add to results)
    - ...and so on.
- Add 3: [3], remaining: 4
    - ...

Try This Test Case Yourself:

candidates = [3,4,5], target = 8

What combinations sum to 8?

Take a moment to sketch or code the solution yourself before checking the answer above!

Reflect:

Did you know this problem can also be solved using dynamic programming? Try it after you're comfortable with backtracking!

Problem 2: Combination Sum II ([Leetcode 40](https://leetcode.com/problems/combination-sum-ii/))

How is This Different?

Now, each number in candidates can be used **at most once**, and candidates may contain duplicates. The goal: find all unique combinations that sum to the target.

Key Twist:

- Each number is used once.
- Must avoid repeating the same combination (even if presented in different order).

Example Input/Output:

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

Output:

```
[
  [1,1,6],
  [1,2,5],
  [1,7],
  [2,6]
]
```

Brute-Force:

Try every subset, but checking all permutations can lead to duplicates. This would be very inefficient and require a way to deduplicate results.

Optimal Approach: Backtracking with Sorting and Skipping Duplicates

- *Sort* the array first, so duplicates are adjacent.
- During backtracking:
    - For each recursion, start from the next index (i + 1), not i (since each number can be used once).

• If you see the same number as before (and it's not the first candidate at this level), skip it to avoid duplicate combinations.

Step-by-Step Pseudocode:

```
function combinationSum2(candidates, target):
    sort candidates
    results = []
    backtrack(remaining, start, path):
        if remaining == 0:
            add copy of path to results
            return
        if remaining < 0:
            return
        for i from start to len(candidates):
            if i > start and candidates[i] == candidates[i - 1]:
                continue  # Skip duplicates at this recursion level
            add candidates[i] to path
            backtrack(remaining - candidates[i], i + 1, path)  # Move forward
            remove last element from path
    backtrack(target, 0, [])
    return results
```

Example Trace:

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

- After sorting: [1,1,2,5,6,7,10]
- At each level, if two adjacent numbers are the same and you haven't picked the first one, skip to avoid duplicates.

Try This Test Case Yourself:

candidates = [2,5,2,1,2], target = 5

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

Step-by-Step Walkthrough of the Code:

- Sort the array: [1,2,2,2,5]
- Start at 0: pick 1
    - Remaining: 4, next pick from 1
    - Pick 2: [1,2]
        - Remaining: 2, pick from 2
        - Pick 2: [1,2,2] remaining: 0
    - Skip duplicate 2 at index 2 and 3 at this level
- Next, start at 1: pick 2
    - Remaining: 3, pick from 2
    - Pick 2: [2,2], remaining: 1, pick from 3
        - Pick 2: [2,2,2], remaining: -1 (stop)
    - Skip duplicate 2 at index 3

- Pick 5: [2,5], remaining: -2 (stop)
- Pick 5 directly: [5], remaining: 0

Complexity:

- Time: O(2^n), where n = len(candidates), due to backtracking, but pruning and duplicate-skipping helps.
- Space: O(n) for the recursion stack and result storage.

Problem 3: Combination Sum III ([Leetcode 216](https://leetcode.com/problems/combination-sum-iii/))

What's New Here?

Now, you must find all possible combinations of *k* numbers from 1 to 9 that sum to a given target. Each number can be used at most once.

Constraints:

- Only numbers 1 to 9.
- Must pick exactly *k* numbers.
- Each number used once.

Example Input/Output:

Input: k = 3, n = 7

Output: [[1,2,4]]

(1 + 2 + 4 = 7)

Pseudocode and Guidance:

```
function combinationSum3(k, n):
    results = []
    backtrack(remaining, start, path):
        if remaining == 0 and length(path) == k:
            add copy of path to results
            return
        if remaining < 0 or length(path) > k:
            return
        for i from start to 9:
            add i to path
            backtrack(remaining - i, i + 1, path)
            remove last element from path
    backtrack(n, 1, [])
    return results
```

How This Differs from Prior Problems:

- Only numbers 1..9, not arbitrary.
- Must use exactly *k* numbers.
- Each number used once, so always advance start index.

Test Case to Try:

k = 4, n = 15

What combinations of 4 numbers from 1 to 9 sum to 15?

Walkthrough:

- For each number from 1 to 9, try adding to path, recurse.
- Stop if path too long or sum exceeded.
- Only add to results if length == k and sum == target.

Potential Complexity:

- Time: O(C(9, k)), since you're choosing k elements from 9.
- Space: O(k) for the recursion stack, plus results.

Reflect:

Notice the similarities to previous problems, but also how the fixed pool and count change the recursion.

Summary and Next Steps

Today you explored three classic backtracking problems that revolve around **combinations under constraints**. You learned how:

- Backtracking can efficiently generate all possible combinations, pruning impossible branches early.
- To handle unlimited reuse (Problem 1), single-use with duplicates (Problem 2), and fixed-size combinations from a set pool (Problem 3).
- Sorting and skipping duplicates is crucial for unique combinations.
- Adjusting your recursion's starting index is key to controlling whether numbers can be reused.

Common Traps:

- Forgetting to sort and skip duplicates (leads to repeated combinations).
- Not resetting the path properly during backtracking.
- Confusing when to advance the start index.
- Missing the base cases (for sum or path length).

Action List

- Solve all three problems on your own, even the one with code provided.
- For Problems 2 and 3, try implementing from scratch without peeking at the solution.
- Experiment with dynamic programming or iterative approaches for Problem 1.
- Compare your code with others on Leetcode — pay attention to how they handle duplicates and edge cases.
- If you get stuck, don't worry! The more you practice backtracking, the more natural it will feel.

Keep experimenting, and backtrack your way to coding mastery!