Topic Introduction

Today, we're diving into the elegant world of **backtracking** — a versatile technique that powers some of the most common "generate all combinations" problems in coding interviews.

What is Backtracking?

Backtracking is a systematic way to try out different solutions for a problem, especially when the goal is to find *all* possible answers. You build up a solution incrementally, one piece at a time, and whenever you realize the current path can't lead to a valid answer, you "backtrack" — undo the last step and try something else.

How does it work?

Think of it like exploring a maze. Every time you reach a fork, you pick a direction. If you hit a dead end, you retrace your steps and try a different path. Backtracking is the algorithmic version of this process.

When and why is it useful?

Backtracking shines when:

- You need to generate all possible arrangements, combinations, or subsets (power sets).
- You have constraints that must be satisfied (e.g., balanced parentheses).
- The search space is too large to brute-force fully, but you can prune paths early.

A Simple Example (not one of today's problems):

Suppose you want to generate all 2-letter combinations from the set `["a", "b", "c"]`.

Starting with an empty string, you try to add each letter. For each choice, you move forward, add the next letter, and so on. If you reach the desired length, you record the combination.

Output: `["ab", "ac", "ba", "bc", "ca", "cb"]` (assuming no repeats of the same letter in a combination).

Why These 3 Problems?

Today's trio — Subsets, Subsets II, and Generate Parentheses — are united by their demand for generating *all valid combinations* using backtracking.

- **Subsets** focuses on finding all possible subsets of a set.
- **Subsets II** asks the same, but with duplicate elements (and no duplicate subsets in output).
- **Generate Parentheses** challenges us to generate all valid strings of parentheses.

They also progress in difficulty and highlight different flavors of backtracking — from simple, to handling duplicates, to enforcing structural constraints.

Let's start with the classic: *Subsets*.

Problem 1: Subsets

Problem Statement (Rephrased):

Given a list of distinct integers, return all possible subsets (the power set). Each subset can be in any order, but no duplicates.

LeetCode Link

Example:

Input: `nums = [1, 2, 3]`
Output: `[[], [1], [2], [3], [1,2], [1,3], [2,3], [1,2,3]]`

Each subset is unique. Order of subsets and numbers within a subset doesn't matter.

How to Think About It:

Imagine you are at each index of the list, and you ask:
"Do I include this number in my current subset, or skip it?"
This gives us two choices at every step, leading to a binary tree of decisions.

Try it with Pen and Paper:

List out the choices for `[1, 2]`:

- Start with `[]`
- Include 1: `[1]`
    - Include 2: `[1,2]`
    - Skip 2: `[1]`
- Skip 1: `[]`
    - Include 2: `[2]`
    - Skip 2: `[]`

Notice how we revisit `[]` twice (which is fine). Ultimately, all possible combinations appear.

Try this test case:

Input: `nums = [0, 1]`
What are all the subsets?

Brute-force Approach:

Enumerate every possible combination by flipping a coin for each number: include or exclude. For `n` numbers, that's `2^n` possibilities.

- **Time Complexity:** O(2^n * n) (since each subset takes O(n) to copy/construct)
- **Space Complexity:** O(2^n * n) (for storing all subsets)

Optimal Approach: Backtracking

- At each step, decide: include current number or not.
- Move to the next index.
- When you reach the end of the list, record the current subset.
- Backtrack by removing the last number and trying the other path.

Let's implement this in Python:

```
def subsets(nums):
    result = []

    def backtrack(start, path):
        # Add the current subset
        result.append(list(path))
        # Try including each number from 'start' to end
        for i in range(start, len(nums)):
            # Include nums[i] in the subset
            path.append(nums[i])
            # Recurse with the next number
            backtrack(i + 1, path)
            # Backtrack - remove last added number
            path.pop()

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

Time and Space Complexity:

- There are `2^n` subsets. Each subset can take up to O(n) space.
- **Time:** O(2^n * n)
- **Space:** O(2^n * n)

Explanation of Each Part:

- `result`: Stores all generated subsets.
- `backtrack(start, path)`: Recursively explores all combinations starting from index `start`, building up the current subset in `path`.
- At each call, we add a *copy* of `path` to `result`.
- The for-loop tries to include each number (from the current position onwards) in the subset.
- After trying with a number, we remove it (backtrack), so we can try the next candidate.

Trace Example:

Input: `[1,2,3]`

- Start: `[]`
    - Add 1: `[1]`
        - Add 2: `[1,2]`
            - Add 3: `[1,2,3]`
        - Backtrack, remove 3: `[1,2]`
    - Backtrack, remove 2: `[1]`
        - Add 3: `[1,3]`
    - Backtrack, remove 3: `[1]`
- Backtrack, remove 1: `[]`

- Add 2: `[2]`
    - Add 3: `[2,3]`
  - Backtrack, remove 3: `[2]`
- Backtrack, remove 2: `[]`
  - Add 3: `[3]`
- Backtrack, remove 3: `[]`

Result: `[[], [1], [1,2], [1,2,3], [1,3], [2], [2,3], [3]]`

Another Test Case to Try:

Input: `nums = [4,5]`
What subsets do you get? Try drawing the recursion tree!

Take a moment to solve this on your own before jumping into the solution.

Problem 2: Subsets II

How is this different?

- Input list may contain **duplicates** (e.g., `[1,2,2]`).
- Output must not contain duplicate subsets.

LeetCode Link

Example:

Input: `[1, 2, 2]`
Output: `[[], [1], [1,2], [1,2,2], [2], [2,2]]`

Notice `[2,1]` is the same as `[1,2]`, so we only want unique subsets.

Brute-force Approach:

- Generate all possible subsets, then remove duplicates (maybe by converting each to a tuple and using a set).
- **Time Complexity:** Still O(2^n * n), but with extra work to remove duplicates.

Optimal Approach: Backtracking with Duplicate Skipping

- **Key Insight:** Sort the array first. This clusters duplicates together.
- When backtracking, skip adding the same number at the same tree level to avoid duplicate subsets.

Step-by-step Logic:

- Sort the input to group duplicates.
- During backtracking, skip any number that is the same as the previous, *unless* it is the first number at this depth/iteration.

Pseudocode:

```
function subsetsWithDup(nums):
    sort(nums)
```

```
    result = []
    backtrack(start, path):
        add copy of path to result
        for i from start to len(nums):
            if i > start and nums[i] == nums[i-1]:
                continue  // skip duplicates at this level
            path.append(nums[i])
            backtrack(i+1, path)
            path.pop()
    backtrack(0, [])
    return result
```

Example Dry Run:

Input: `[1, 2, 2]` (after sorting: same)

- Start: `[]`
  - Add 1: `[1]`
    - Add 2: `[1,2]`
      - Add 2: `[1,2,2]`
    - Backtrack, remove 2: `[1,2]`
    - At i=2, nums[2]==nums[1], so skip (already handled at i=1)
  - Backtrack, remove 2: `[1]`
- Backtrack, remove 1: `[]`
  - Add 2: `[2]`
    - Add 2: `[2,2]`
  - Backtrack
  - At i=2, nums[2]==nums[1], so skip

Result: `[[], [1], [1,2], [1,2,2], [2], [2,2]]`

Another Test Case to Try:

Input: `[2,2,2]`
What unique subsets do you get?

Time and Space Complexity:

- **Time:** O(2^n * n) (number of subsets, each up to n in length)
- **Space:** O(2^n * n) (for all subsets)

Trace:

Try running the above pseudocode on `[1,2,2]` and see how duplicates are skipped.

Problem 3: Generate Parentheses

What's new here?

- Instead of subsets, we must generate all **valid** combinations of n pairs of parentheses.
- Constraints: At any point, you can't have more closing than opening parentheses.

LeetCode Link

Example:

Input: `n = 3`
Output: `["((()))","(()())","(())()","()(())","()()()"]`

Brute-force Approach:

- Generate all possible strings of `2n` parentheses, and filter those that are valid.
- **Inefficient:** The majority of strings will be invalid.

Optimal Approach: Backtracking with Constraints

- Track counts of open and close parentheses used so far.
- Only add an open parenthesis if you have not used up all n.
- Only add a close parenthesis if open > close (i.e., you have an unmatched open to close).

Pseudocode:

```
function generateParenthesis(n):
    result = []
    function backtrack(open, close, path):
        if len(path) == 2 * n:
            add path to result
            return
        if open < n:
            backtrack(open+1, close, path + "(")
        if close < open:
            backtrack(open, close+1, path + ")")
    backtrack(0, 0, "")
    return result
```

Example Trace:

For `n = 2`, valid combinations are: `"()()"`, `"(())"`

- Start: `open=0, close=0, path=""`
    - Add "(": `open=1, close=0, path="("`
        - Add "(": `open=2, close=0, path="(("`
            - Can't add more "(", so add ")": `open=2, close=1, path="()"`
                - Add ")": `open=2, close=2, path="(())" [done]`
        - Backtrack to `open=1, close=0, path="("`
            - Add ")": `open=1, close=1, path="()"`
                - Add "(": `open=2, close=1, path="()("`

• Add ")": `open=2, close=2, path="()()" [done]`

Another Test Case for You:

Input: `n = 1`
What valid combinations are there?

Time and Space Complexity:

- Number of valid combinations is the nth Catalan number (~4^n / n^1.5).
- Each combination is length 2n.
- **Time:** O(4^n / sqrt(n))
- **Space:** O(4^n / sqrt(n) * n)

Give it a try!

Try implementing this using the above pseudocode. Test with `n = 3` and `n = 2`. Can you see how the backtracking avoids invalid combinations?

Summary and Next Steps

Why These Problems?

- They're all classic "generate all combinations" questions.
- They showcase backtracking: building up solutions incrementally, and backtracking when a path is complete or invalid.
- Each problem adds a twist:
    - Subsets: All combinations.
    - Subsets II: Handle duplicates.
    - Parentheses: Enforce constraints.

Key Insights:

- Always try to prune invalid or duplicate paths early in backtracking to save time.
- Sorting can help when handling duplicates (as in Subsets II).
- For problems with validity rules (like parentheses), encode the constraints directly into your recursion.

Common Mistakes:

- Not making a *copy* of the current subset/path when adding to results.
- Forgetting to backtrack (remove the last element) after recursion.
- Not sorting or skipping duplicates (leading to duplicate subsets).
- For parentheses: Allowing too many closing parentheses, leading to invalid strings.

Action List

- Solve all 3 problems on your own — even if you read the code above.
- Try solving Subsets II and Generate Parentheses with iterative or other approaches.
- Explore related problems, like Permutations, Combination Sum, or N-Queens.
- Compare your solution with others. How do you handle edge cases? Is your recursion clean?

- If you get stuck, review the recursion trees with pen and paper. Visualizing the process is powerful!
- Remember: Backtracking is a skill. Practice will make it second nature.

You've just leveled up your backtracking skills! Keep at it.