Topic Introduction

Welcome to today's PrepLetter! Today, we're diving into a classic coding interview technique: **Backtracking for String Generation**.

What is backtracking?
Backtracking is a powerful algorithmic technique used to build solutions incrementally, one piece at a time, and remove solutions that fail to satisfy constraints as soon as possible (this is called "pruning"). It's like exploring a maze: at every intersection, you try a path; if you hit a dead end, you back up and try a different route.

How does it work?
You make a choice, move forward, and, if you realize the choice doesn't lead to a solution, you undo (backtrack) and try another option. This is often implemented using recursion, where each recursive call explores a new possibility.

When is it useful?
Backtracking shines in problems where you need to generate all possible combinations or arrangements, especially when some combinations are invalid. Interviewers love it because it tests your understanding of recursion, constraints, and pruning for efficiency.

Simple Example (Not one of today's problems):
Suppose you want to generate all possible 2-letter combinations from the set {A, B, C}.
You would start with an empty string, then for each position, try adding A, B, or C, and recursively build the next letter. If you reach length 2, you add the combination to your answer.

Why is this useful in interviews?
Backtracking is a versatile pattern appearing in puzzles, games, constraint satisfaction, and string generation. It tests your ability to design recursive solutions, prune impossible paths, and think about both correctness and performance.

Today, we'll explore three problems that all use backtracking to **generate strings** under specific rules:

- Letter Combinations of a Phone Number
- Generate Parentheses
- Restore IP Addresses

Why are these grouped together?
All three ask you to systematically try out different ways to build a string, only keeping those that satisfy certain constraints. Phone numbers become letter combinations, parentheses must be valid, and digit strings must become valid IP addresses. The core pattern is the same, but the constraints differ!

Let's start exploring!

Problem 1: Letter Combinations of a Phone Number

Link:
https://leetcode.com/problems/letter-combinations-of-a-phone-number/

Problem Rephrased:

Given a string of digits (from '2' to '9'), return all possible letter combinations that the digits could represent, using the mapping of digits to letters on a phone keypad.

Example:

Input: `"23"`

Output: `["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"]`

(Here, '2' maps to 'a', 'b', 'c'; '3' maps to 'd', 'e', 'f')

Step-by-Step Thought Process:

- Each digit corresponds to a set of letters.
- For each digit in the input, for each letter it maps to, build up combinations with every letter of the next digit, and so on.
- If you have `n` digits, you want all possible strings of length `n` where each character is one of the possible mapped letters for that digit.

Pen-and-paper tip:

Try drawing a decision tree for `"23"` to see how the choices branch out.

Additional test case to try:

Input: `"7"`

Output: `["p", "q", "r", "s"]`

('7' maps to four letters: 'p', 'q', 'r', 's')

Brute-force approach:

- Use nested loops for each digit, but since the number of digits isn't fixed, hardcoding loops is not scalable.
- Time complexity: O(3^n), where n is the number of digits (since each digit maps to up to 4 letters).

Optimal approach: Recursive Backtracking!

- At each position, try every letter for the current digit, add it to your current string, and move to the next digit.
- When you've built a string as long as the input, add it to your results.

Python Solution

```python
def letterCombinations(digits):
    # Mapping from digit to possible letters
    phone_map = {
        '2': 'abc', '3': 'def', '4': 'ghi', '5': 'jkl',
        '6': 'mno', '7': 'pqrs', '8': 'tuv', '9': 'wxyz'
    }
    result = []

    # Edge case: empty input
    if not digits:
        return result

    # Helper function for backtracking
    def backtrack(index, path):
        # If the path is as long as the digits, we have a combination
        if index == len(digits):
            result.append(path)
            return
        # For each letter mapped to the current digit
        for letter in phone_map[digits[index]]:
            backtrack(index + 1, path + letter)

    backtrack(0, "")
    return result
```

Time Complexity: O(3^n * n), where n is the length of `digits` (each digit maps to up to 4 letters, and we build strings of length n).
Space Complexity: O(3^n * n) for result storage and recursion stack.

Code Breakdown:

- `phone_map`: Maps each digit to its possible letters.
- `result`: Stores all valid combinations.
- `backtrack(index, path)`:
    - `index`: The current position in the input digits.
    - `path`: The string built so far.
    - When `index` equals the length of `digits`, we've built a complete combination.
    - For the current digit, try every possible letter, add it to `path`, and recurse to the next digit.
- The recursion explores every possible path, building up letter combinations.

Trace for Input `"23"`:

- Start at index 0, path `""`
    - Try 'a' (`"a"`)
        - Try 'd' (`"ad"`) -> done (add to result)
        - Try 'e' (`"ae"`) -> done
        - Try 'f' (`"af"`) -> done
    - Try 'b'...
    - Try 'c'...

Try this input yourself:
Input: `"29"`
What are all the possible combinations? (Tip: '2' maps to 'a', 'b', 'c'; '9' maps to 'w', 'x', 'y', 'z')

Your turn:
Take a moment to solve this on your own before jumping into the solution. Try drawing the recursion tree for a short input!

Problem 2: Generate Parentheses

Link:

https://leetcode.com/problems/generate-parentheses/

Problem Rephrased:

Given `n` pairs of parentheses, generate all combinations of well-formed (valid) parentheses strings.

Example:

Input: `n = 3`

Output: `["((()))", "(()())", "(())()", "()(())", "()()()"]`

Why is this similar to the previous problem?

Again, we're generating all possible strings under certain constraints, using backtracking. The key difference: **not all arrangements are valid** — we must ensure that at no point do we close more parentheses than we've opened.

Brute-force:

- Generate all possible strings of length `2n` using '(' and ')', then filter those that are valid.
- But this is highly inefficient: there are 2^(2n) possible strings, but very few are valid.

Optimal Approach: Backtracking with Constraints

- At every step, you can add '(' if you haven't used up all `n` opening brackets.
- You can add ')' only if it won't cause more ')' than '(' so far (i.e., only if you have more opens than closes).
- When you've used all n opens and n closes, add the combination to your result.

Pseudocode

```
function generateParenthesis(n):
    result = []
    backtrack(open=0, close=0, path="")
    return result

function backtrack(open, close, path):
    if length of path == 2 * n:
        add path to result
        return
    if open < n:
        backtrack(open + 1, close, path + '(')
    if close < open:
        backtrack(open, close + 1, path + ')')
```

Example trace for n = 2:

- Start with ""
    - Add '(': "("
        - Add '(': "(("
            - Add ')': "(()"
                - Add ')': "(())" (done)
            - (can't add '(' anymore)

            - Add ')': "()"
                - Add '(': "()(("
                    - Add ')': "()()" (done)

Test case for you:

Input: `n = 1`

What are the valid outputs?

Step-by-step explanation:

- At each step, try to place an open parenthesis if possible.
- Only place a close parenthesis if it wouldn't make the string invalid (i.e., there are more opens than closes).
- Build the string recursively until it's length 2n.

Time Complexity:

The number of valid parentheses strings for `n` is the nth Catalan number: Cn ~ 4^n / (n^1.5 * sqrt(pi)). So, time complexity is O(4^n / sqrt(n)), with each string being length 2n.

Space Complexity:

O(4^n / sqrt(n)), proportional to the number of valid strings.

Try this input yourself:

Input: `n = 2`

What are all valid combinations? See if you can list them before coding!

Problem 3: Restore IP Addresses

Link:

https://leetcode.com/problems/restore-ip-addresses/

Problem Rephrased:

Given a string containing only digits, return all possible valid IP address combinations by inserting three dots to divide the string into four valid octets (each 0-255, no leading zeros unless the octet is '0').

What's different or more challenging?

Now, instead of just picking from a fixed mapping or following a simple rule (open/close), you must:

- Decide *where* to place the dots (splitting positions).
- Ensure each segment is valid (not "00" or "256").
- The input string can be of varying length.

Brute-force:

- Try every way to split the string into four parts, then check if each is a valid octet. There are up to O(n^3) ways to split, but many are invalid.

Optimal Approach: Backtracking with Constraints and Choice of Split Points

- At each step, try to take 1, 2, or 3 digits as the next octet (if possible).
- Only proceed if the segment is a valid IP octet (0-255, no leading zeros unless '0').
- Recurse for the next part, keeping track of how many octets are left.

- When 4 octets have been placed and no digits remain, add the built IP to the result.

Pseudocode

```
function restoreIPAddresses(s):
    result = []
    backtrack(start=0, path=[], dots=0)
    return result

function backtrack(start, path, dots):
    if dots == 4 and start == length of s:
        result.append(join path with '.')
        return
    if dots == 4 or start == length of s:
        return
    for length in 1 to 3:
        if start + length > length of s:
            break
        segment = s[start:start+length]
        if isValid(segment):
            backtrack(start + length, path + [segment], dots + 1)

function isValid(segment):
    if segment starts with '0' and length > 1:
        return False
    if int(segment) > 255:
        return False
    return True
```

Example for s = "25525511135":

- Try "255.255.11.135" (valid)
- Try "255.255.111.35" (valid)
- Try other combinations, but prune if segment > 255 or has leading zeros.

Test case for you:

Input: `"101023"`

What valid IP addresses can you form?

Time Complexity:

O(1), since the string can be at most 12 digits (4 octets * 3 digits). If unbounded, O(n^3), since there are up to O(n^3) ways to place 3 dots.

Space Complexity:

O(1) or O(number of valid IPs), again because the total number of results is small for bounded input.

Reflective prompt:

How would you handle this for IPv6 (hexadecimal, different segment lengths)? What's the general backtracking pattern here?

Summary and Next Steps

Today, you tackled three string-generation problems where **backtracking** is the hero:

- Build possibilities step-by-step
- Prune invalid partial solutions early
- Recursively explore all valid answers

Key patterns to remember:

- For each decision (letter, bracket, segment), try all valid choices.
- Use constraints to avoid unnecessary work (prune!).
- Backtrack by undoing choices as you return from recursion.

Common pitfalls:

- Forgetting base cases (e.g., when to add a solution).
- Not handling constraints correctly (e.g., leading zeros in IPs, invalid parentheses).
- Stack overflow from infinite recursion (always move toward base case).

Action List

- Solve all 3 problems on your own, even the one with code provided.
- For Problem 2 and 3, try writing a solution using iterative approaches, or with memoization where possible.
- Explore other backtracking problems (e.g., permutations, combinations, N-Queens).
- Compare your solutions with others, especially for clarity and handling edge cases.
- Practice drawing recursion trees and decision diagrams to internalize backtracking flows.
- If you get stuck, review your base cases and constraint checks — and keep practicing!

Keep at it! Each problem you tackle makes recursion and backtracking a little more intuitive and a lot more fun.