Knowledge Review

Last time, we introduced the basic idea of **backtracking**: it's a strategy built upon recursion to explore multiple possible paths.

The core of recursion is to break down a large problem into smaller subproblems of the same structure, until reaching the simplest "base cases" that can be directly solved.

Backtracking is a special kind of recursion, but it focuses more on the idea that **each level of recursion represents a decision point**, and uses recursion to explore the consequences of each choice.

In other words, every recursive call is a decision made for the current path. You can think of it this way: at any given state, you're standing at a fork in the road. Each option is a *choice*, and making that choice leads to a new subproblem — which leads to the next layer of recursion.

The essence of backtracking is: try every path (every choice), once you find that it is not feasible, **undo the previous choice, return to the original state, and try another path**. So, backtracking not only includes the process of breaking down the problem recursively, but also systematically explore the entire space of possible decisions.

Backtracking Template: The Three Key Questions

To solve backtracking problems, I summarize the solution into a **“Three-Question Template”**:

1.When do we stop?

This step defines the *termination condition* of the recursion — the "leaf node" of the decision tree. This condition tells us whether a path is complete and whether we should stop searching further.

Common stop conditions include:

- the path has reached a certain length,
- or we've traversed all elements,
- or we've met a specific constraint given by the problem.

2.When do we store the result?

In backtracking, we don’t store results at every step — we only add a path to the result set when it meets all problem requirements.

This usually happens with the stop condition.

For example:

- only store combinations of length *k*,
- only record valid parentheses,
- or only collect fully filled chessboards.

3.How do we choose and turn back?

This is the **core** of backtracking. We must clearly understand:

- what choices we can make at the current step (i.e., the search space),
- how we go deeper into the recursion after making a choice,

- and how we "undo" the choice after the recursion returns, restore the previous state and try other options.

The third question can be broken down into **three sub-questions**:

> - **How many choices can we choose?** What are the candidate options at this point? What is the search space? Where do we start iterating?
> - **How to go into the next recursion after choosing?** After making a choice, how do we move to the next recursive layer? Do we need to update the path, mark the current element as used, move a pointer, or adjust any variables?
> - **How to revoke the choice (turn back)?** After returning from recursion, how do we restore the state so we can try the next choice? Do we remove the last element from the path? Do we unmark the used status?

This is the 'three-question' template. After this, let's look at how we can adjust this template based on different types of backtracking problems, by analyzing the characteristics of the five most common categories.

Question Type Summary

The five most common categories is:

- Subset problems
- Permutation problems
- Combination problems
- String segmentation problems
- Board problems

This is a summary table.

	Subset	**Permutation**	**Combination**	**String segmentation**	**Board**
Question Key Information	Find all valid subsets	Find all permutations of elements	Find all valid combinations of k elements summing to target	Given a string, find all valid ways to split it	Place elements or search paths on a 2D board/grid under constraints
Analysis	- Usually no strong constraints - Each element is either chosen or not - Order doesn't matter	- Each element used once - Order matters - All elements are selected eventually	- No repetition - Order doesn't matter - May include sum/length constraints	- Each substring must satisfy given rules (e.g., in dictionary / palindrome / valid IP) - Splits must be contiguous - May include constraints on count, length, or validity	- Positions must not conflict (e.g., queens/sudoku) - Paths may not revisit; limited directions
State definition	`path` + current `startIndex`	`path` + `visited[]` flag array	`path` + current `startIndex`	`path` + current starting index (`startIndex`)	Board state (`board`) + current level (e.g., row number or position)
When do we stop?	When `startIndex` reaches the end of array	When `path.length` equals input array length	When `path.length` satisfies given condition (e.g., equals k)	When `startIndex` reaches end of string	When all elements are placed or board is fully filled
When do we store?	Every step can potentially add a result (if valid)	Store when `path.length` equals input array length	Store when path meets conditions (e.g., length == k)	Store when we reach the end and all segments are valid	Store when a valid final state is reached (e.g., N queens placed)
How many choices can	Remaining elements from	All unused elements	Remaining elements from	All substrings starting from `startIndex` to the	All valid positions or values at current

we choose?	`startIndex` onward		`startIndex` onward	end	state
How to go into the next recursion after choosing?	Add current element, increment `startIndex + 1`	Add to path and mark as used	Add current element, `startIndex + 1`	Add substring to path and move `startIndex` to split end	Modify board state (e.g., place queen), go to next row/position
How to revoke the choice (turn back)?	Remove last element from `path`	Remove last element + reset `visited[i]` to false	Remove last element from `path`	Remove last substring, restore `startIndex`	Remove element from board and restore board state

Now that we've gone through the differences between the 5 common backtracking problem types, let's practice with a real example.

Let’s start with LeetCode 216: Combination Sum III.

Practice

216. Combination Sum III

I will give you 2 minutes to read this question.

After reading the question, we can extract the following key information:

- Selection range: integers from 1 to 9
- Selection restrictions: each element is used at most once and cannot be repeated; the order is not important

- Valid result is combined with k non-repeating elements and the sum is n

Then, let's start thinking about how to start writing code according to the Three Key Questions template.

```
class Solution(object):
    def combinationSum3(self, k, n):
        """
        :type k: int
        :type n: int
        :rtype: List[List[int]]
        """
        # Analysis the problem and get key infomation
        '''
        - Selection range: integers from 1 to 9
        - Selection restrictions: each element is used at most once and
cannot be repeated; the order is not important
        - Valid result rule: k non-repeating elements constitute and the
sum is n
        '''

        # initilize variable to store valid path
        self.result = []

        # define the recursion function
        def dfs(startIndex, path, remain, number):
            '''
            1. When do we stop?
            2. When do we store the result?
            3. How do we choose and turn back?
              - How many choices can we choose?
              - How to go into the next recursion after choosing?
              - How to revoke the choice (turn back)?
```

```
                '''
                if number == 0:
                    if remain == 0:
                        self.result.append(path[:])
                    return

                for i in range(startIndex, 10):
                    path.append(i)
                    remain -= i
                    dfs(i+1, path, remain, number-1)
                    path.pop()
                    remain += i

            dfs(1, [], n, k)
            return self.result
```

First, let's think about the question **"When do we stop?"**

As we mentioned earlier, a valid result must consist of exactly k elements, so once the number of selected elements reaches **k**, we should stop.

To check this, we need a variable to track how many elements we've selected so far. I choose to use a variable called `number`, which is initially set to **k**, and we decrease it by 1 every time we select a number. When `number` reaches 0, we know it's time to stop — and that solves the stopping condition.

Next, let's think about **"When do we store the result?"**

We know that a valid result must not only have k elements, but also sum up to n. So once we reach the stopping condition, we need to check whether the sum of the selected numbers equals n. If it does, we store the current path as a valid result.

To make this easier, I use a variable called `remain` to represent how much value we still need to reach the target sum. If `remain` is exactly 0 when we stop, it means the current combination sums to **n**, and that's the time to record it.

Now comes the core part: **"How do we choose and turn back?"** Let's break it down into smaller questions and handle them one by one.

First, we need to know what the current search space is.

From the problem description, we're only allowed to pick numbers from 1 to 9, so that gives us the range to work with. But if we always start choosing from 1 in each recursion, we'll end up generating multiple permutations of the same set of numbers. And according to the problem, the order doesn't matter, but we can't have duplicates. To avoid duplicates, we need a mechanism to make sure that we're only choosing forward, not backward. That's why we introduce a variable called `startIndex` — it ensures that each level of recursion only chooses numbers after the current one. This naturally gives us a for-loop structure that starts from `startIndex`.

Next, how do we go into the next recursion after choosing a number?

After choosing a number, we need to update both the number of remaining selections (`number`) and the remaining target sum (`remain`). Since we've used one number, `number` should decrease by 1, and `remain` should decrease by the value we just picked. Also, `startIndex` should move forward so that we don't re-select the same or smaller numbers. This gives us the recursive call.

Finally, we think about **"How do we revoke the choice (turn back)?"**

This is just the reverse of what we did at previous step. We remove the last number from the path, and reset `number` and `remain` to their previous values. Just like this.

And that's how we finish writing the backtracking function. All it's left is to call the function and return the result. Let's give it a try.

That’s all for my part today. I hope this walkthrough helps you better understand how to solve backtracking problems. Next is **Timmons** — let’s hand over the screen to him.