# Topic Introduction

Today, we're diving into a fascinating corner of string manipulation: **string pattern matching**. This is more than just searching for a word inside another word. Instead, it’s about figuring out if two sequences (strings, words, or even lists) follow the same **structural pattern**.

**What does "pattern matching" mean here?**
Suppose you have two strings, "egg" and "add." They look different, but if you replace each unique letter in the first string with a unique symbol, can you transform one into the other? That’s pattern matching — checking if the structure or mapping of elements is the same.

**How does it work?**
Pattern matching problems often rely on mapping elements from one sequence to another. The most common approach is to use a dictionary (or hash map) to keep track of these mappings. You check if each new element fits the mapping established by previous elements.

**Why is this useful in interviews?**
Pattern matching tests your ability to recognize structural similarities, work with hash maps, and manage constraints — all key skills for roles involving algorithms and data structures.

**A simple example (not from our problems):**
Suppose you have a pattern "abba" and a word sequence "dog cat cat dog."
Assign "a" to "dog" and "b" to "cat." The pattern is satisfied.
But "abba" and "dog cat cat fish"? The mapping breaks, so it’s not a match.

Now, let’s look at three popular LeetCode problems using this concept:

- **Isomorphic Strings:** Check if two strings can be mapped to each other with a one-to-one character correspondence.
- **Word Pattern:** Determine if a sequence of words follows the same pattern as a given string of pattern characters.
- **Group Shifted Strings:** Group strings that are "shifted" versions of each other, i.e., the pattern of letter changes is the same.

Why group them?
All three are about **identifying and matching patterns** within or between strings, using mapping as the core technique. The first two focus on direct mapping, while the third asks you to discover and group by a shift-based structural pattern.

Let’s jump in!

## Problem 1: Isomorphic Strings

Isomorphic Strings - LeetCode #205

**Rephrased Problem:**
Given two strings `s` and `t`, determine if you can replace each character in `s` with a unique character to get `t`, such that the mapping is consistent and one-to-one.

**Example:**

`s = "egg"`, `t = "add"`

- Map 'e' to 'a', both 'g's to 'd'.
- So "egg" becomes "add" — isomorphic!

**Output:** `True`

But:

`s = "foo"`, `t = "bar"`

- 'f' to 'b', 'o' to 'a' (first 'o'), but second 'o' should also map to 'a', but in 'bar' it's 'r' — mapping fails.

**Output:** `False`

**Try this test case yourself:**

`s = "paper"`, `t = "title"`

**Brute-force Approach:**

Try all possible mappings from `s` to `t` and see if any work.

- This is exponential time and not practical.

**Optimal Approach:**

Use two hash maps:

- One to map from `s` to `t`.
- Another to check that no two letters in `s` map to the same letter in `t` (injectivity).

**Step-by-step logic:**

- For each character in `s` and its corresponding character in `t`:
    - If the character in `s` is already mapped, check if it maps to the current character in `t`. If not, return False.
    - If not mapped, make sure the character in `t` is not already mapped to by another character in `s`. If not, create the mapping.
- If you reach the end without conflict, return True.

**Python Solution:**

```
def isIsomorphic(s: str, t: str) -> bool:
    # Two dictionaries to keep track of the mappings
    s_to_t = {}
    t_to_s = {}

    for char_s, char_t in zip(s, t):
        # Check if there is a previous mapping from s to t
        if char_s in s_to_t:
            # If the mapping doesn't match the current character, return False
            if s_to_t[char_s] != char_t:
                return False
        else:
            # If char_t is already mapped from another char_s, return False
            if char_t in t_to_s:
                return False
            # Otherwise, create the mapping
```

```
            s_to_t[char_s] = char_t
            t_to_s[char_t] = char_s

    return True
```

**Time Complexity:** O(n), where n is the length of the strings (since you scan both once).
**Space Complexity:** O(n), for the hash maps.

**Trace Example:**
`s = "paper"`, `t = "title"`

- p -> t
- a -> i
- p -> t (already mapped, OK)
- e -> l
- r -> e

All mappings are consistent and one-to-one.
**Output:** `True`

**Try this test case:**
`s = "ab"`, `t = "aa"`
What does your code return? (Hint: Should be False — both 'a' and 'b' can't map to 'a'.)

**What does each part do?**

- The two dictionaries keep track of mappings in both directions to ensure each character only maps to one character and vice versa.
- The zip function pairs up corresponding characters from `s` and `t`.
- We check mappings at every step to avoid conflicts.

**Take a moment:**
Try implementing this yourself — or dry-run the code with the test case above!

**Extra Reflection:**
Did you know you could also solve this by encoding both strings into their first-occurrence patterns (e.g., "egg" -> [0,1,1]) and comparing the results? Give that a try after this!

## Problem 2: Word Pattern

Word Pattern - LeetCode #290

**Rephrased Problem:**
Given a pattern string (like "abba") and a space-separated string of words (like "dog cat cat dog"), check if the word sequence follows the same letter pattern.

**How is this similar to Problem 1?**

- Both require a one-to-one mapping.

- Here, pattern characters must uniquely map to words.

**Example:**

`pattern = "abba"`, `s = "dog cat cat dog"`

- 'a' -> "dog", 'b' -> "cat", and the sequence follows the pattern.

**Output:** `True`

But:

`pattern = "abba"`, `s = "dog cat cat fish"`

- Last word is "fish" instead of "dog" — mapping breaks.

**Output:** `False`

**Try this test case yourself:**

`pattern = "abc"`, `s = "dog dog dog"`

**Brute-force Approach:**

Try all possible assignments from pattern letters to words.

- This is impractical for large inputs.

**Optimal Approach:**

Use two hash maps:

- One from pattern characters to words.
- One from words to pattern characters (to avoid two pattern chars mapping to the same word).

**Pseudocode:**

```
split words from s
if length of pattern != number of words:
    return False

for each (pat_char, word) pair:
    if pat_char previously mapped:
        if mapping != word:
            return False
    else:
        if word already mapped to another pat_char:
            return False
        map pat_char to word, and word to pat_char

return True
```

**Step-by-step:**

- Split `s` into words.
- If the number of pattern characters doesn't match the number of words, return False.
- For each pair:
    - Check mapping consistency as in Problem 1.

**Example trace:**

`pattern = "abba", s = "dog cat cat dog"`

- a -> "dog"
- b -> "cat"
- b -> "cat" (OK)
- a -> "dog" (OK)

**Output:** `True`

**Try this test case:**
`pattern = "aaaa", s = "dog cat cat dog"`
Should return False.

**Time Complexity:** O(n), where n is the number of words.

**Space Complexity:** O(n), for hash maps.

## Problem 3: Group Shifted Strings

Group Shifted Strings - LeetCode #249

**Rephrased Problem:**
Given a list of strings, group all strings that are "shifted" versions of each other.
A shifted string is formed by shifting each letter by the same amount (e.g., "abc" -> "bcd").

**What's different here?**
Instead of direct mapping, we need to identify a "shift pattern" for each string. Strings with the same shift pattern go in the same group.

**Example:**
Input: `["abc", "bcd", "acef", "xyz", "az", "ba", "a", "z"]`

- "abc", "bcd", "xyz" are all shifted versions (shift between letters is [1,1])
- "az", "ba" (shift is [25])
- "acef" is alone (shift [2,2])
- "a", "z" (single letters form their own group)

**Output:**
`[["abc","bcd","xyz"],["az","ba"],["acef"],["a","z"]]`

**Try this test case:**
Input: `["a", "b", "c", "d"]`
(Should all be grouped together.)

**Optimal Approach:**
Create a **signature** for each string based on the difference between consecutive characters, modulo 26 (for wrap-around). Use this signature as a key to group strings.

**Pseudocode:**

```
initialize groups = empty map

for each string in list:
    if length is 1:
        signature = ()
    else:
        for i in 1 to len(string)-1:
            diff = (ord(string[i]) - ord(string[i-1]) + 26) % 26
            add diff to signature
        signature = tuple of diffs

    add string to groups[signature]

return list of groups' values
```

**Example trace:**

For "abc":

- 'b' - 'a' = 1, 'c' - 'b' = 1 => signature (1,1)

For "bcd":

- 'c' - 'b' = 1, 'd' - 'c' = 1 => signature (1,1)

They go in the same group.

**Try this test case:**

Input: `["ace", "bdf", "egi", "adf"]`

Work out the signatures and groupings.

**Time Complexity:** O(n * k), where n is the number of strings, and k is the average length of a string.

**Space Complexity:** O(n * k), for storing the groups and signatures.

# Summary and Next Steps

**What did we learn?**

All three problems are about **identifying and matching patterns in strings**:

- Using hash maps to track one-to-one mappings.
- Creating unique signatures to group by structural similarity.
- Being careful to enforce bijection (no two keys to same value, and vice versa).

**Key insights:**

- Always check both directions in mapping problems.
- For grouping, a "signature" function is crucial.
- Watch for wrap-around in alphabet-based problems.

**Common mistakes:**

- Forgetting to check the mapping in both directions.
- Not accounting for word/character count mismatch.
- Overcomplicating the signature in group-based problems.

**Your Action List:**

- Solve all 3 problems on your own — even the one with code provided above.
- For Group Shifted Strings, try both the signature approach and your own ideas.
- Explore related problems such as "Pattern Matching", "Anagrams", or "Bijective Mapping" on LeetCode.
- Compare your code with others — pay attention to edge cases and code readability.
- If you get stuck, review the mapping technique and dry-run with pen and paper.

Keep practicing — the more patterns you master, the stronger your algorithmic intuition will become!