Topic Introduction

Today, we're diving into the world of **string searching and pattern matching** — a classic interview theme that shows up in many forms. Whether you're looking for a substring, checking if a string repeats itself, or hunting for efficient ways to match patterns, mastering these techniques will pay off big-time in coding interviews.

What is string searching?
String searching is the process of finding one string (the "pattern" or "needle") within another string (the "text" or "haystack"). Sometimes, it's as simple as finding the first occurrence. Other times, it's about checking if the entire string is built by repeating a smaller substring, or finding all occurrence indices efficiently.

How does it work?
At its core, you're comparing parts of the string (slices, substrings) to see if they match another string. This can be as basic as a naive scan, or as advanced as using algorithms that optimize comparisons (like KMP).

When and why is this useful in interviews?
String searching is fundamental in many real-world apps: spellcheckers, search boxes, DNA analysis, plagiarism detection, and more. Interviewers love these problems because they reveal your approach to brute-force, optimization, and edge cases.

Let's look at a tiny, simple example:
Suppose you want to find the word "apple" in the sentence "I have an apple pie."
You'd start at each position in the sentence and see if the next five letters match "apple". If yes, you've found it!

Now, let's tackle three LeetCode problems that are all about this core idea — but each with its own twist.

Why are these three problems grouped together?

All three involve **string searching and pattern matching**:

- **strStr()**: Find the index of the first occurrence of a substring in another string.
- **Repeated Substring Pattern**: Check if a string can be constructed by repeating a substring.
- **KMP String Matching**: Implement an efficient substring search algorithm (the classic Knuth-Morris-Pratt/strStr).

They build on each other: from simple searching, to pattern awareness, to efficient matching.

Problem 1: Implement strStr()

LeetCode Problem

Problem Statement (Rephrased):
Given two strings, `haystack` and `needle`, return the index of the first occurrence of `needle` in `haystack`. If `needle` is not part of `haystack`, return -1.

Example:
Input:

```
haystack = "hello"
```

```
needle = "ll"
```

Output:

```
2
```

Explanation: The substring "ll" starts at index 2 in "hello".

Another Test Case to Try:

Input:

```
haystack = "abcdef"
needle = "gh"
```

Expected Output:

```
-1
```

Thinking Through the Solution:

If you’re stuck, grab a pen and write down both strings, then scan the haystack one character at a time, checking if the next few letters match the needle.

Brute-force Approach:

Check every substring of `haystack` with the same length as `needle` and compare character by character.

- Time Complexity: O(N*M), where N is the length of haystack and M is the length of needle.
- Space Complexity: O(1)

Optimal Approach:

For this problem, brute-force is perfectly acceptable because the constraints are friendly.
But, in real interviews, you may be asked about more efficient approaches (like KMP), which we’ll explore in Problem 3.

Step-by-step Logic:

- Loop through each position `i` in `haystack` from 0 to `len(haystack) - len(needle)`.
- For each `i`, check if the substring `haystack[i:i+len(needle)]` equals `needle`.
- If yes, return `i`.
- If the loop completes with no match, return -1.

Python Solution:

```
def strStr(haystack: str, needle: str) -> int:
    """
    Returns the index of the first occurrence of needle in haystack, or -1 if not found.
    """
    # Edge case: empty needle should return 0 by convention
    if needle == "":
        return 0

    n, m = len(haystack), len(needle)
    # Loop through haystack, but only up to where needle could fit
    for i in range(n - m + 1):
        # If the substring matches needle, return current index
        if haystack[i:i + m] == needle:
            return i
    # If loop completes, needle not found
    return -1
```

Time Complexity: O(N*M)

Space Complexity: O(1) (no extra space besides variables)

Explanation:

- The function first handles the case where `needle` is empty (by problem definition, return 0).
- It then loops through each possible starting index in `haystack`, extracting a substring of length `len(needle)` and checking for a match.
- If found, returns the index; otherwise, returns -1 if no match is found.

Trace Example:

Let's dry-run `haystack = "mississippi"`, `needle = "iss"`:

- i = 0: "mis" != "iss"
- i = 1: "iss" == "iss" → Match! Return 1

Try this test case yourself:

haystack = "banana", needle = "na"

What index will it return?

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

Problem 2: Repeated Substring Pattern

LeetCode Problem

Problem Statement (Rephrased):

Given a non-empty string `s`, check if it can be constructed by repeating a substring multiple times.

Example:

Input:

```
s = "abab"
```

Output:

```
True
```

Explanation: "ab" repeated twice makes "abab".

Another Test Case to Try:

Input:

```
s = "abcabcabcabc"
```

Expected Output:

```
True
```

How is this similar or different?

Like strStr, we're looking for patterns in a string. But instead of searching for a single occurrence, we check if the whole string is made of repeating a substring.

Brute-force Approach:

Try all possible substring lengths from 1 to len(s)//2. For each length, check if repeating that substring enough times makes the original string.

- Time Complexity: O(N^2), where N is the length of the string.
- Space Complexity: O(N) for building the repeated string.

Optimal Approach:

We can use string manipulation and some clever observations (or the KMP prefix table, as in the next problem). But let's stick with the simple optimal check:

Step-by-step Logic:

- For each possible substring length `l` from 1 to len(s)//2:
  - If len(s) is divisible by `l`, get substring `s[0:l]`.
  - Repeat this substring `len(s)//l` times.
  - If it matches `s`, return True.
- If no such substring found, return False.

Pseudocode:

```
function repeatedSubstringPattern(s):
    n = length of s
    for l from 1 to n//2:
        if n % l == 0:
            pattern = s[0:l]
            if pattern repeated (n // l) times == s:
                return True
    return False
```

Trace Example:

For s = "abcabcabcabc":

- Try l=1: "a" * 12 != s
- Try l=2: "ab" * 6 != s
- Try l=3: "abc" * 4 == s → return True

Try this test case:

s = "abcdabcdabce"

What does the function return?

Time Complexity: O(N^2) in the worst case (many substring checks), but usually fast in practice.

Space Complexity: O(N) when building the repeated string.

Problem 3: KMP String Matching (Efficient strStr)

LeetCode Problem

Problem Statement (Rephrased):
Implement a substring search (like strStr), but efficiently — using the Knuth-Morris-Pratt (KMP) algorithm, which avoids unnecessary comparisons.

What's new or more challenging here?
Unlike the previous brute-force search, KMP pre-processes the pattern to build a "prefix table" (also called lps: longest proper prefix which is also suffix). This lets you skip ahead smartly when you find mismatches, leading to linear time searching.

How is this related to the previous problems?
It's the same task as strStr, but now you're asked to do it efficiently, applying the clever KMP pattern.

Pseudocode:
```
function buildLPS(pattern):
    lps = array of zeroes, length = len(pattern)
    length = 0
    for i from 1 to len(pattern) - 1:
        while length > 0 and pattern[i] != pattern[length]:
            length = lps[length - 1]
        if pattern[i] == pattern[length]:
            length += 1
            lps[i] = length
    return lps

function KMP_search(text, pattern):
    if pattern is empty:
        return 0
    lps = buildLPS(pattern)
    i = 0, j = 0
    while i < len(text):
        if text[i] == pattern[j]:
            i += 1
            j += 1
            if j == len(pattern):
                return i - j
        else:
            if j != 0:
                j = lps[j - 1]
            else:
                i += 1
    return -1
```

Trace Example:
text = "aabaaabaaac", pattern = "aabaaac"

- Build LPS for "aabaaac": [0,1,0,1,2,2,0]

- Walk through text, using lps to skip unnecessary checks.

Try this test case:

text = "mississippi", pattern = "issip"

What index should be returned?

Potential Time Complexity:

- Preprocessing: O(M) for building the lps table (M = length of pattern)
- Searching: O(N) (N = length of text)
- Overall: O(N + M)

Space Complexity: O(M) for the lps array.

Reflect:

Can you think of a case where the brute-force approach would be much slower than KMP? Try constructing one!

Summary and Next Steps

We explored three classic problems that all center on **string searching and pattern matching**. From a straightforward substring search, to checking repeated patterns, and finally to building an efficient KMP matcher, you've seen how simple ideas can scale up to handle much bigger strings efficiently.

Key Patterns and Insights:

- Always start with brute-force to clarify the problem.
- Look for repeated work: can you pre-process or reuse comparisons?
- KMP's lps table is a powerful tool for skipping ahead without missing matches.
- Pattern matching isn't just about finding, but about how you find efficiently.

Common Mistakes:

- Off-by-one errors in substring slices.
- Forgetting edge cases: empty strings, pattern longer than text.
- Not resetting indices appropriately in KMP.

Action List:

- **Solve all three problems on your own**, even the one with the code provided.
- Try solving Problem 2 and 3 using a different technique (for example, rolling hash or using built-in methods, or even try to code KMP without looking at the pseudocode).
- Explore other string searching problems (like Rabin-Karp, or finding all occurrences instead of just the first).
- Compare your solution with others — especially for edge-case handling and code style.
- Don't worry if you get stuck! The most important thing is to keep practicing and learning from each attempt.

Happy string searching!