The KMP Algorithm: Intelligent String Matching

KMP 算法：智能字符串匹配

A deep dive into the linear-time pattern matching algorithm by Knuth, Morris, and Pratt.

深入解析由 Knuth、Morris 和 Pratt 提出的线性时间模式匹配算法。

The Challenge: Finding a Pattern in Text
挑战：在文本中寻找模式

String searching is a core problem: Given a text `t` of length `n` and a pattern `p` of length `m`, find all occurrences of `p` within `t`.

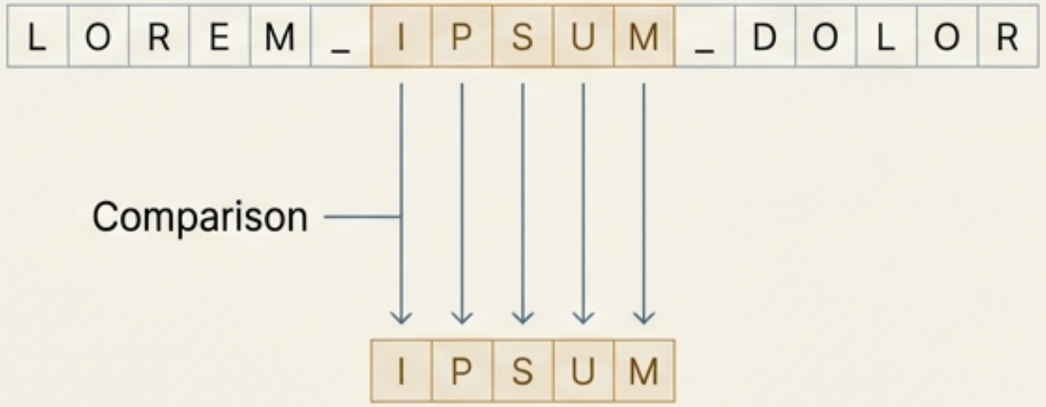
字符串搜索是一个核心问题：给定一个长度为 `n` 的文本 `t` 和一个长度为 `m` 的模式 `p`，找到 `p` 在 `t` 中的所有出现位置。

The most straightforward method is the Naive Algorithm. It checks for a match at every possible starting position in the text.

最直接的方法是朴素算法。它在文本的每一个可能起始位置检查是否匹配。

When a mismatch occurs, it simply shifts the pattern one position to the right and starts the comparison all over again from the beginning of the pattern.

当发生不匹配时，它只是将模式向右移动一个位置，然后从模式的开头重新开始比较。

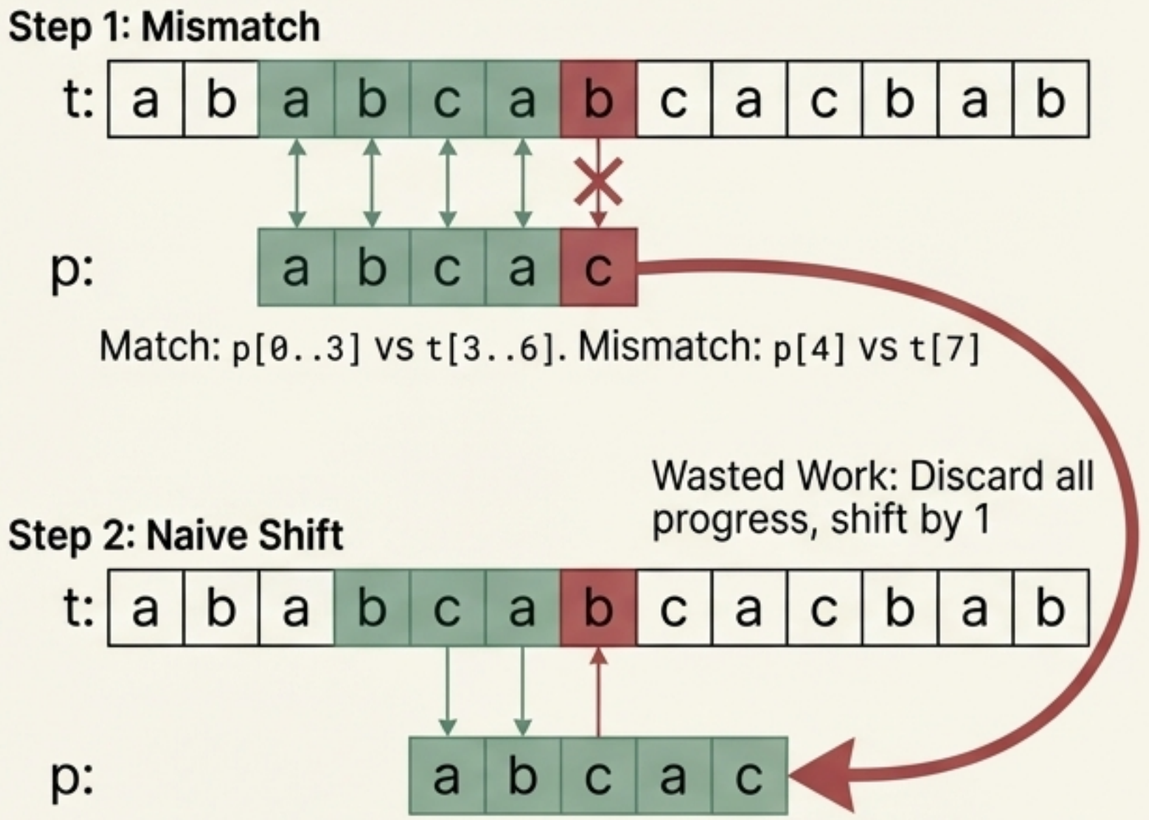


The Inefficiency of the Naive Approach
朴素算法的低效性

Consider this mismatch scenario. After matching `p[0..3]` with `t[3..6]`, we find a mismatch at `p[4]`.
思考图 **9-2(c)** 中的这个不匹配场景。在 `p[0..3]` 与 `t[3..6]` 匹配后，我们在 `p[4]` 处发现不匹配。

The naive algorithm discards all information from the partial match. It shifts the pattern by one position (`i=4`) and restarts the comparison from `j=0`.
朴素算法丢弃了部分匹配中的所有信息。它将模式移动一个位置（`i=4`）并从 `j=0` 重新开始比较。

This leads to many redundant comparisons, resulting in a **worst-case** time complexity of O((n-m)m).
这导致了许多冗余的比较，其最坏情况下的时间复杂度为 O((n-m)m)。

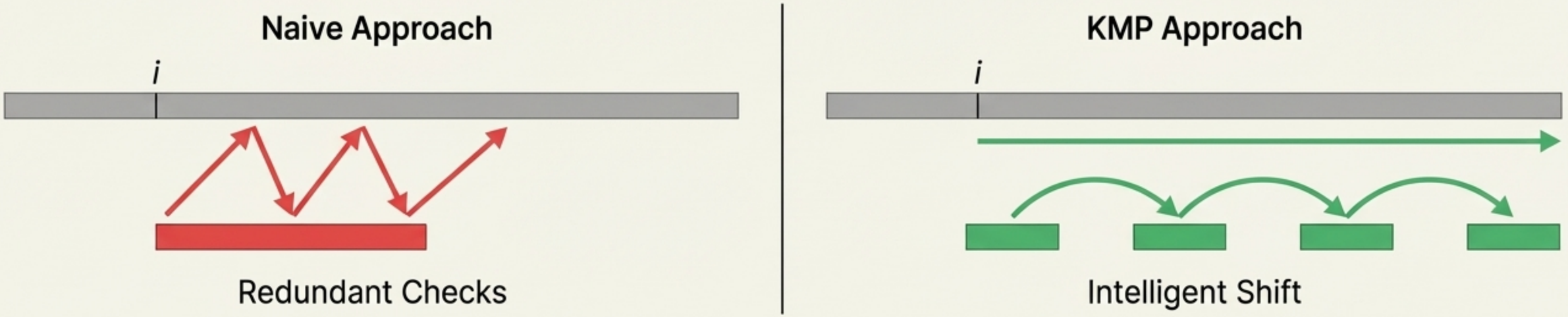


The KMP Insight: Don’t Forget the Past
KMP 的洞见：利用已知信息

The KMP algorithm, developed by Knuth, Morris, and Pratt, dramatically improves performance to O(m+n).
由 Knuth、Morris 和 Pratt 开发的 KMP 算法，将性能显著提升至 O(m+n)。

The Key Idea": When a mismatch occurs, we use information from the already-matched prefix of the pattern to determine the furthest safe position to shift the pattern, avoiding redundant comparisons.
核心思想[1][2]：当不匹配发生时，我们利用已经匹配的模式前缀信息来确定模式可以安全移动到的最远位置，从而避免冗余比较。

Crucially, the KMP algorithm never moves the text pointer backward.
关键在于，KMP 算法从不回溯文本指针。

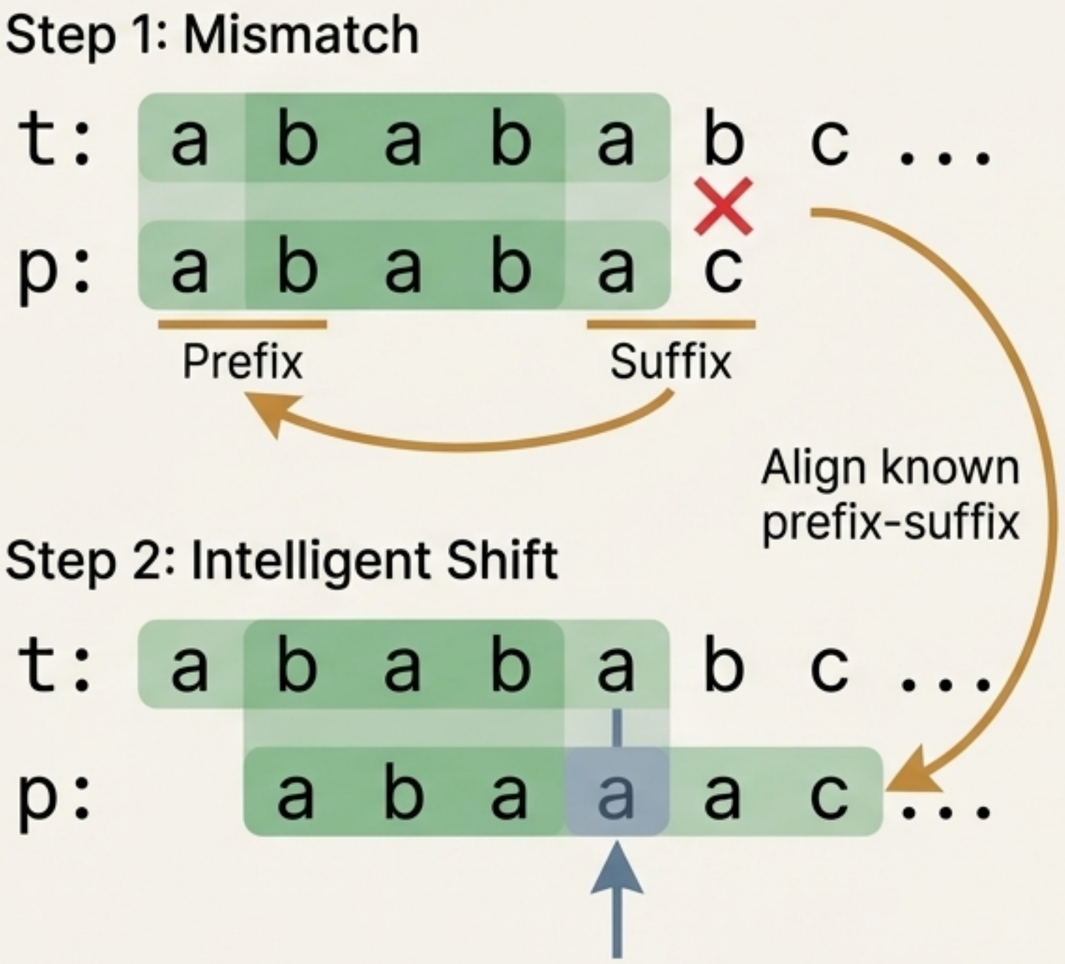


The KMP Shift in Action
KMP 算法的智能移动

Let's revisit a mismatch. In this example, `t[i+q]` does not match `p[q]`.
让我们重新审视一个不匹配的例子。在此场景中，`t[i+q]` 与 `p[q]` 不匹配。

Instead of shifting by one, KMP analyzes the matched prefix `p[0..q-1]`. It knows that the prefix `p[0..1]` ('ab') also appears as a suffix of the matched part.
KMP 并非简单地移动一位，而是分析已匹配的前缀 `p[0..q-1]`。它知道前缀 `p[0..1]` ('ab') 同样作为已匹配部分的后缀出现。

It intelligently shifts the pattern to align this prefix-suffix, and resumes the comparison from `p[2]`. No characters in the text are re-compared.
它智能地移动模式来对齐这个前缀-后缀，并从 `p[2]` 继续比较。文本中的字符没有被重复比较。

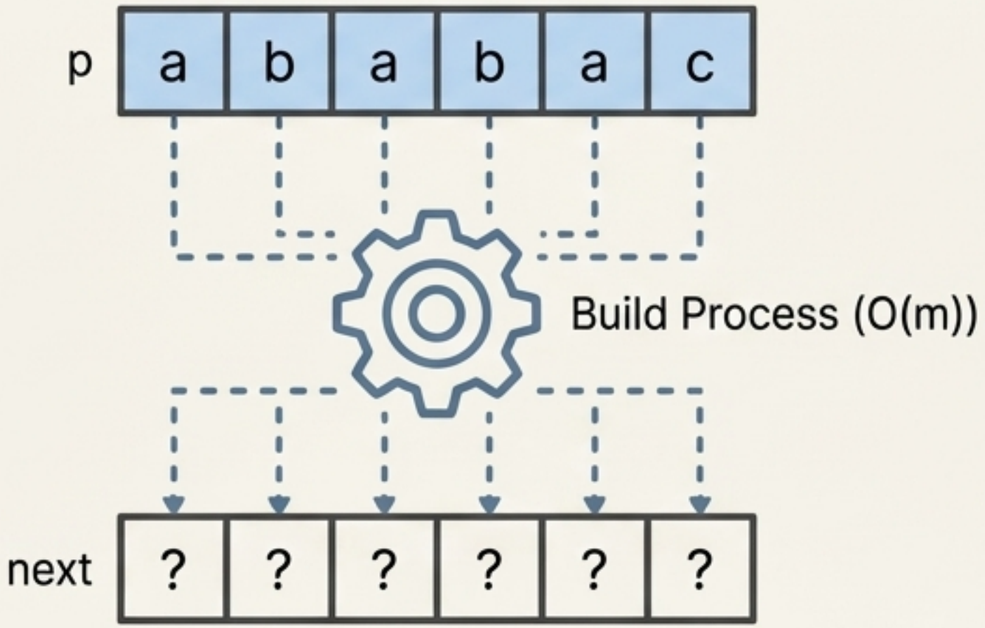


The Engine: The `next` Array
核心引擎：`next` 数组

KMP's intelligence is not magic; it's pre-computation. The algorithm first builds an auxiliary array, `next`, based solely on the pattern `p`.
KMP 的智能并非魔法，而是预计算。该算法首先仅根据模式 `p` 构建一个辅助数组 `next`。

Purpose: For any position `q` in the pattern, `next[q]` stores information about the longest proper prefix of `p[0..q]` that is also a suffix of `p[0..q]`.
目的：对于模式中的任意位置 `q`，`next[q]` 存储了 `p[0..q]` 的最长真前缀的信息，该真前缀同时也是 `p[0..q]` 的后缀。

This array acts as a lookup table. When a mismatch occurs at `p[q]`, we consult `next[q-1]` to know how far to shift the pattern.
这个数组就像一个查找表。当在 `p[q]` 处发生不匹配时，我们查询 `next[q-1]` 来得知应该移动多远。



Formal Definition of the `next` Array
`next` 数组的正式定义

For a pattern `p[0..m-1]`, the `next` array is defined for each position `q` (from 0 to m-1). The source code implements a slightly adjusted version for 0-based indexing and loop convenience, where `next[q]` stores the index `j` of the last character of the prefix.
对于一个模式 `p[0..m-1]`，`next` 数组为每个位置 `q`（从 0 到 m-1）进行定义。源代码实现了一个为 0-索引和循环便利性一稍作调整的版本，其中 `next[q]` 存储的是前缀最后一个字符的索引 `j`。

The conceptual definition from the source is (Equation 9.2): 源码中的概念性定义为（公式 9.2）：

$$next(q) = \max \{k \mid p[0..k] \text{ is a suffix of } p[0..q]\}$$

(Note: This implies `k < q` for a non-trivial shift, making `p[0..k]` a proper prefix)
（注意：这意味着 `k < q` 以实现有意意义的移动，使得 `p[0..k]` 成为一个真前缀）

Code-Aligned Definition
In the provided C++ code, `next[i] = j` means the longest proper prefix of `p[0..i]` that is also a suffix has length `j+1`. `j` is the index of the last character of that prefix. `next[0]` is set to -1 as a base case.
在提供的 C++ 代码中，`next[i] = j` 意味着 `p[0..i]` 的最长真前缀（同时也是后缀）的长度为 `j+1`。`j` 是该前前缀最后一个字符的索引。`next[0]` 被设为 -1 作为基准情况。

Example: Building `next` for 'abababababca'
示例：为 'abababababca' 构建 `next` 数组

q	0	1	2	3	4	5	6	7	8	9
p[q]	a	b	a	b	a	b	a	b	c	a
next[q]	-1	-1	0	1	2	3	4	5	-1	0

q=2, p[0..2]='aba': The longest proper prefix that is also a suffix is 'a' (`p[0]`). Length is 1. `next[2] stores the index `0`.

q=2, p[0..2]='aba'：最长的既是真前缀又是后缀的子串是 'a' (`p[0]`) 。长度为 1。`next[2] 存储索引 `0`.

q=6, p[0..6]='abababa': The longest proper prefix that is also a suffix is 'ababa' (`p[0..4]`). Length is 5. `next[6] stores the index `4`.

q=6, p[0..6]='abababa'：最长的既是真前缀又是后缀的子串是 'ababa' (`p[0..4]`) 。长度为 5。`next[6] 存储索引 `4`.

q=8, p[0..8]='abababababc': No proper prefix is also a suffix. The value is -1 (indicating a full reset).

q=8, p[0..8]='abababababc'：没有任何真前缀同时也是后缀。值为 -1（表示完全重置）。

The KMP Matching Logic
KMP 匹配逻辑

The `KMP-Matcher` function iterates through the text `t` with pointer `i` and the pattern `p` with pointer `j`. At each step `t[i]`, there are three possibilities:

`KMP-Matcher` 函数使用指针 `i` 遍历文本 `t`，使用指针 `j` 遍历模式 `p`。在 `t[i]` 的每一步，有三种可能性：

1. Match: `p[j+1] == t[i]`

The characters match. We advance both pointers (`i++`, `j++`) to check the next characters.

字符匹配。我们同时推进两个指针 (`i++`, `j++`) 来检查下一对字符。

2. Mismatch & Shift: `p[j+1] != t[i]` and `j > -1`

A mismatch occurs. Instead of incrementing `i`, we consult the `next` array. We update `j` to `next[j]`, effectively shifting the pattern to the next best alignment, and repeat the comparison for `t[i]`.

发生不匹配。我们不增加 `i`，而是查询 `next` 数组。我们将 `j` 更新为 `next[j]`，这相当于将模式移动到下一个最佳对齐位置，并对 `t[i]` 重复比较。

3. Mismatch at Start: `p[j+1] != t[i]` and `j == -1`

A mismatch occurs at the very beginning of the pattern. We cannot shift further. We simply advance the text pointer (`i++`) and keep `j` at -1.

在模式的最开始就发生不匹配。我们无法再移动。我们只需推进文本指针 (`i++`) 并保持 `j` 为 -1。

The `KMP-Matcher` Implementation

`KMP-Matcher` 代码实现

```
int KMP-Matcher(const string& t, const string& p) { // KMP 匹配器
    n = t.length();
    m = p.length();
    build(p, next); // Pre-compute the next array (预计算 next 数组)

    int j = -1; // j is the index of the last char in the matched prefix (j 是已匹配前缀的最后一个字符的索引)
    for (int i = 0; i < n; i++) {
        // While mismatch, find the next best prefix using the next array
        // 当不匹配时，使用 next 数组找到下一个最佳前缀
        while (j > -1 && p[j+1] != t[i])
            j = next[j];

        // If characters match, extend the current match
        // 如果字符匹配，扩展当前匹配
        if (p[j+1] == t[i])
            j++;

        // If the entire pattern is matched
        // 如果整个模式都已匹配
        if (j == m-1)
            return i-m+1; // Return the starting index of the match (返回匹配的起始索引)
    }
    return -1; // Not found (未找到)
}
```

Case 2 & 3: Mismatch

The core of the intelligent shift. If a mismatch occurs, `j` is updated using the `next` **array** to find the next possible alignment without moving `i` backwards.

Case 1: Match

A character matches. The length of the current matched prefix is extended by incrementing `j`.

Success

The entire pattern has been found. The function returns the starting index of the match in the text.

Pre-computation: The `build` Function
预计算：`build` 函数

The `build` function is responsible for creating the `next` array.

`build` 函数负责创建 `next` 数组。

The Insight: The logic for building the `next` array is nearly identical to the matching algorithm itself. It efficiently computes the longest prefix-suffix information by "matching the pattern against itself."

核心洞见： 构建 `next` 数组的逻辑与匹配算法本身几乎完全相同。它通过“将模式与自身进行匹配”来高效地计算最长的前缀-后缀信息。

It iterates through the pattern, calculating `next[i]` using the already computed values for `next[0]...next[i-1]`.

它遍历模式，利用已经计算出的 `next[0]...next[i-1]` 的值来计算 `next[i]`。

The `build` Function Implementation
`build` 函数代码实现

```
void build(const string& p, int *next) { // Builds the next array (构建 next 数组)
    int m = p.length();
    next[0] = -1;
    int j = -1;
    for (int i = 1; i < m; i++) {
        // Find length of the longest proper prefix of p[0..i-1] that is a suffix
        // 寻找 p[0..i-1] 的最长真前缀，该前缀同时也是其后缀
        while (j > -1 && p[j+1] != p[i])
            j = next[j];

        // If the next character extends
        // 如果下一个字符扩展了前缀-后缀匹配
        if (p[j+1] == p[i])
            j++;

        // Store the length (as index j)
        // 存储长度（以索引 j 的形式）
        next[i] = j;
    }
}
```

Notice the structural similarity to the `KMP-Matcher` loop. This is the 'self-matching' in action. The pattern `p` is being compared against itself (`p[i]` vs `p[j+1]`) to find the boundaries of its own internal prefixes that are also suffixes.

注意其结构与 `KMP-Matcher` 循环的相似性。这就是 '自我匹配' 的体现。模式 `p` 正在与自身（`p[i]` 对 `p[j+1]`）进行比较，以找到其内部既是前缀又是后缀的子串边界。

A Linear-Time Solution
线性时间解决方案

Column 1: `build` function (Preprocessing)
预计算：`build` 函数

The `for` loop runs `m` times. The work inside the `while` loop can be amortized. The total time complexity is **O(m)**.

`for` 循环运行 `m` 次。`while` 循环内部的工作可以被摊销。总时间复杂度为 **O(m)**。

Column 2: `KMP-Matcher` function (Matching)
匹配：`KMP-Matcher` 函数

The `for` loop runs `n` times. The text pointer `i` never moves back. The pattern pointer `j` can move back, but the total number of 'regressions' via `j = next[j]` is bounded by the number of advances. The total time complexity is **O(n)**.

`for` 循环运行 `n` 次。文本指针 `i` 从不后退。模式指针 `j` 可以后退，但通过 `j = next[j]` 的“回退”总次数受限于前进的总次数。总时间复杂度为 **O(n)**。

Total KMP Time Complexity = O(m) + O(n) = O(m+n)

KMP 总时间复杂度 = O(m) + O(n) = **O(m+n)**

This is a significant asymptotic improvement over the naive O((n-m)m) algorithm, especially for long patterns and texts.

这相对于朴素算法的 O((n-m)m) 是一个显著的渐进改进，尤其是在处理长模式和长文本时。

Key Takeaways: The KMP Philosophy
核心要点：KMP 的设计哲学

Preprocessing is Power: The O(m) investment in building the `next` array pays off by enabling an O(n) search.

预处理就是力量：在构建 `next` 数组上 O(m) 的投入，换来了 O(n) 的搜索效率。

Learn from Mismatches: A **mismatch** provides valuable information about the text. KMP leverages this by consulting the pattern's **internal structure** (`next` array) to avoid fruitless comparisons.

从不匹配中学习：一次不匹配提供了关于文本的宝贵信息。KMP 利用这一点，通过查询模式的内部结构（`next` 数组）来避免无效比较。

Never Look Back (in the Text): The text pointer `i` **only ever increments**, guaranteeing a **linear-time** scan of the text.

永不回头（在文本中）：文本指针 `i` 只会递增，保证了对文本的线性时间扫描。

The Elegance of KMP

KMP 算法的优雅之处

The Knuth-Morris-Pratt algorithm is a classic example of **algorithmic elegance**. By transforming a brute-force problem through clever **pre-computation**, it reveals a **fundamental principle**: **understanding the structure** of your problem's components—in this case, the **pattern itself**—is the key to unlocking dramatic gains in efficiency.

Knuth-Morris-Pratt 算法是算法设计优雅性的经典范例。通过巧妙的预计算，它将一个暴力求解的问题转化，揭示了一个基本原则：深刻理解问题组成部分的结构——在此即模式本身——是实现效率巨大提升的关键。