String Matching Algorithms: The Naive Approach

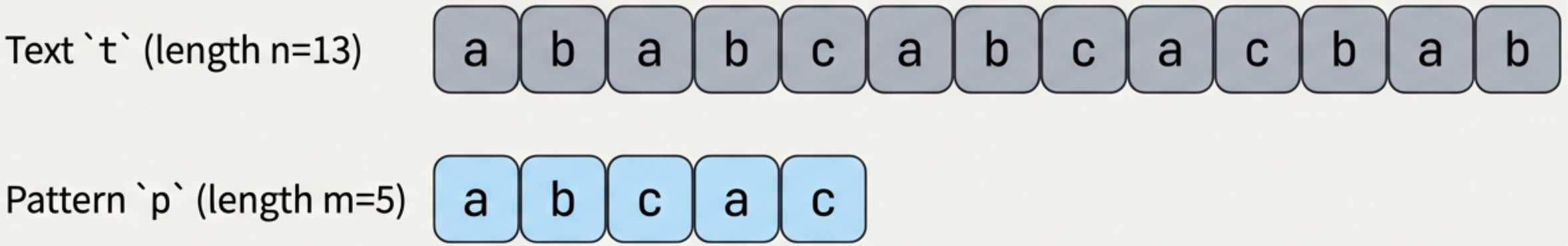
字符串匹配算法：朴素解法

The String Matching Problem

字符串匹配问题

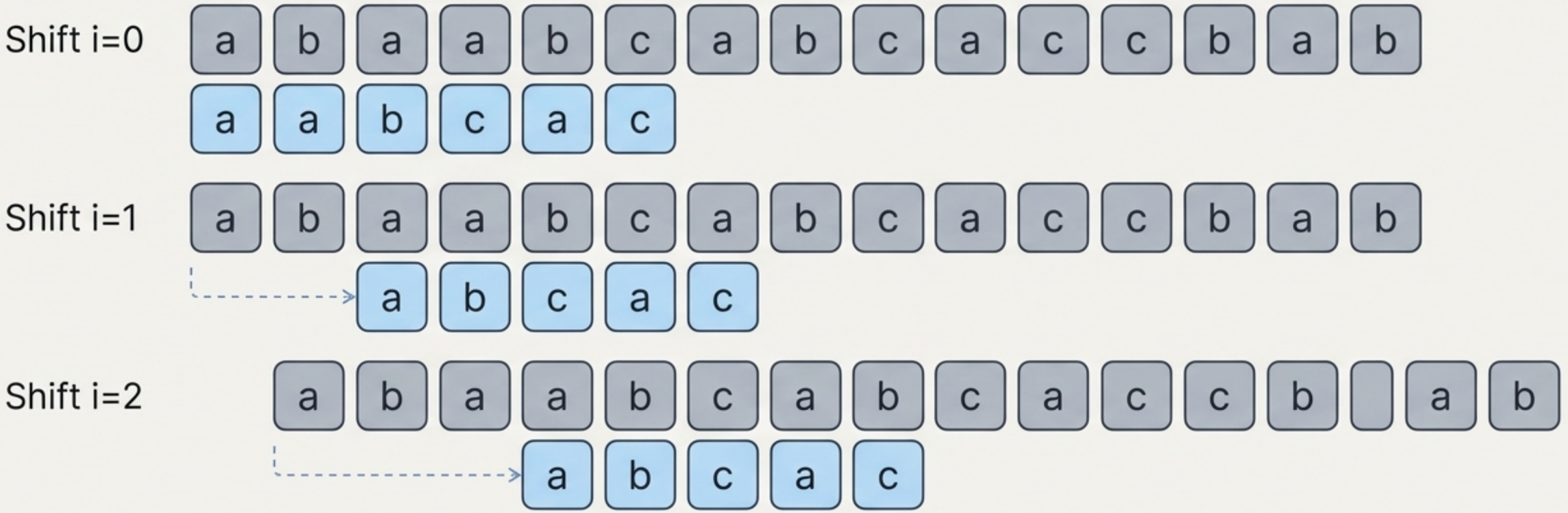
Given a text string `t` of length `n` and a pattern string `p` of length `m` (where m < n), find the starting index of all valid occurrences of `p` within `t`.

给定一个长度为n的主串 t[0..n-1] 和一个长度为m的模式串 p[0..m-1]（m<n），寻找 p 在 t 中所有出现位置的起始索引。



The Intuitive Strategy: Slide and Compare
朴素策略：滑动并比较

The most straightforward method is to slide the pattern *p* across the text *t*, one position at a time. At each position, we perform a character-by-character comparison.
最直接的方法是将模式串p在文本串t上逐位滑动。在每个位置，我们都进行逐字符的比较。



The `naive` Algorithm Implementation

`naive` 算法实现

```
// 朴素的模式匹配算法
int naive(const string& t, const string& p) {
    int n = t.length();
    int m = p.length();
    int i = 0;
    while(i <= n - m) {
        int j = 0;
        while(j < m && t[i+j] == p[j])
            j++;
        if(j == m)
            return i;
        i++;
    }
    return -1;
}
```

Code Deconstructed: The Outer Loop (The "Slide")
代码解析：外层循环（滑动）

```
int naive(const string& t, const string& p) {
    int n = t.length();
    int m = p.length();
    int i = 0;
    while(i <= n - m) { // This loop controls the "slide"
        ...
        i++;
    }
    return -1;
}
```

The variable `i` represents the "shift"—the starting position in text `t` where we attempt to match the pattern. It iterates from `0` to the last possible starting point, `n-m`.

变量`i`代表“移位”——即在文本串`t`中尝试匹配模式串的起始位置。它的迭代范围是从`0`到`n-m`。

Code Deconstructed: The Inner Loop (The "Compare")
代码解析：内层循环（比较）

```
...
while(i <= n - m) {
    int j = 0;
    while(j < m && t[i+j] == p[j])
        j++;
    if(j == m)
        return i; // A match is found!
    ...
}
...
```

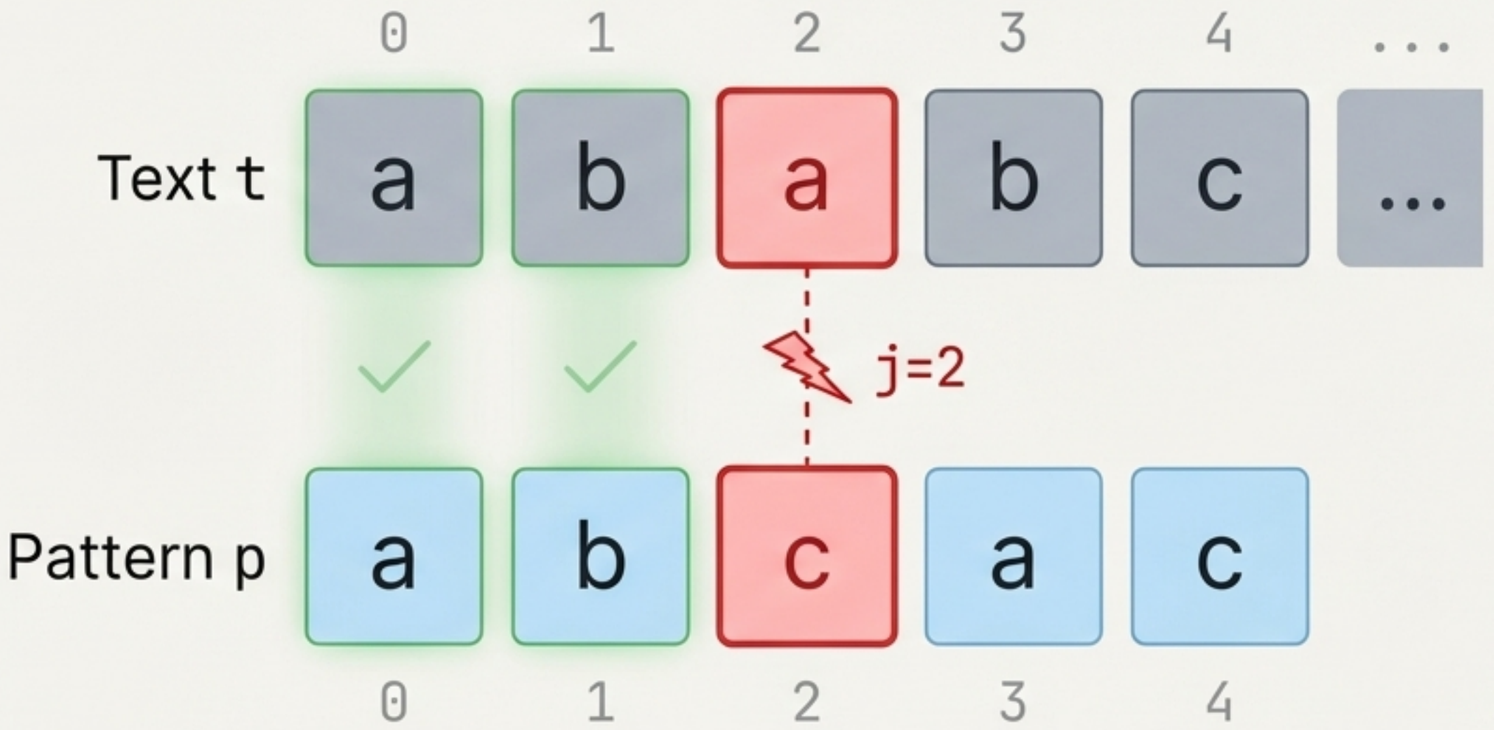
For each shift `i`, the variable `j` iterates from `0` to `m-1`, comparing `p[j]` with `t[i+j]`. If the loop completes without a mismatch (`j` reaches `m`), we have found a full match.

对于每一次移位`i`，变量`j`从`0`迭代到`m-1`，比较`p[j]`与`t[i+j]`。如果循环完整执行（j到达`m`），我们就找到了一个完全匹配。

Algorithm in Action: An Attempt at i=0

算法实战：在 i=0 处的尝试

`t = "ababcabcacbab", p = "abcac"`

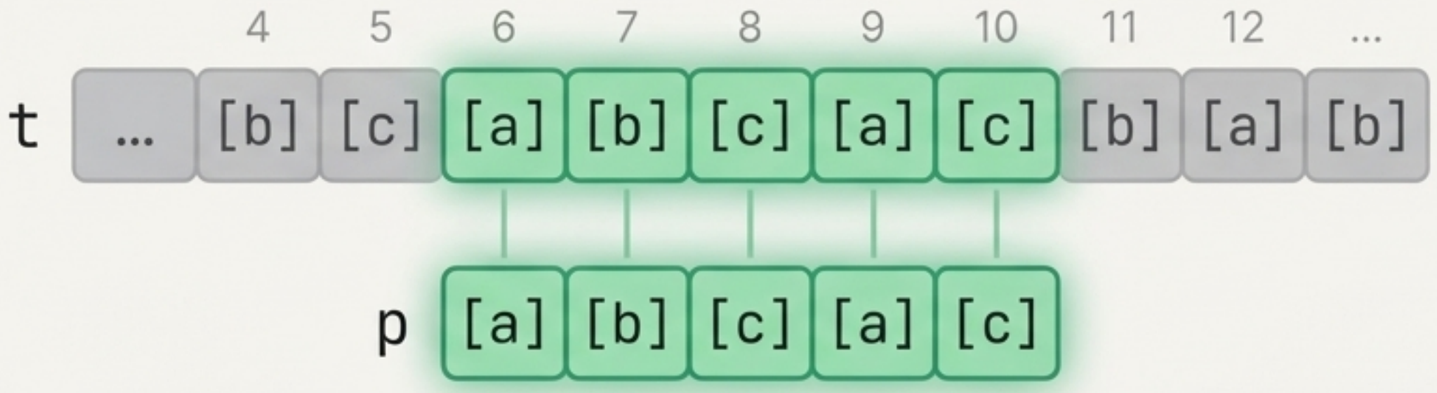


At shift i=0, the comparison proceeds. A mismatch occurs at j=2 because `t[2]` ('a') is not equal to `p[2]` ('c'). The inner loop terminates. i is incremented to 1 for the next attempt.

在移位 i=0 时，索引 j=2 处出现不匹配（因为 t[2] 的 'a' != p[2] 的 'c'）。内层循环终止。我们增加 i 并尝试下一个位置。

Algorithm in Action: Success at `i=6`
算法实战：在 `i=6` 处匹配成功

After several unsuccessful shifts, the algorithm tests the alignment at `i=6`.

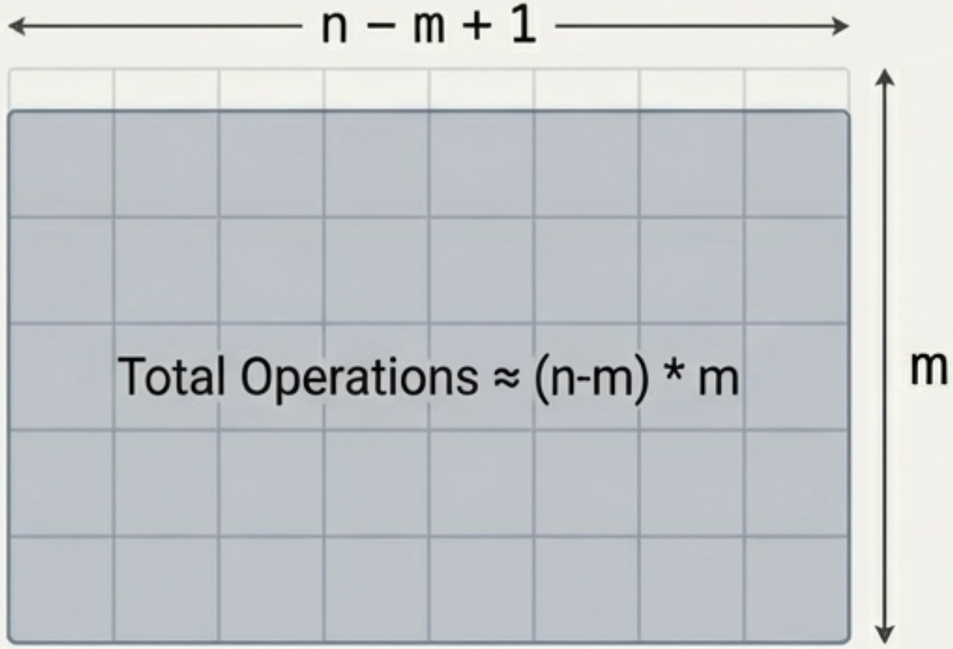


At shift `i=6`, the substring `t[6...10]` is `abcac`. Every character matches `p`. The inner loop runs to completion, and `j` becomes 5 (which equals `m`). The condition `j == m` is true, and the algorithm returns the starting index `i=6`.

在移位 `i=6` 时，子串 `t[6...10]` 为 `abcac`，与 `p` 的每个字符都匹配。内层循环完整执行，`j` 变为 5（`m` 的值）。匹配成功，算法返回索引 `i=6`。

Performance Analysis
性能分析

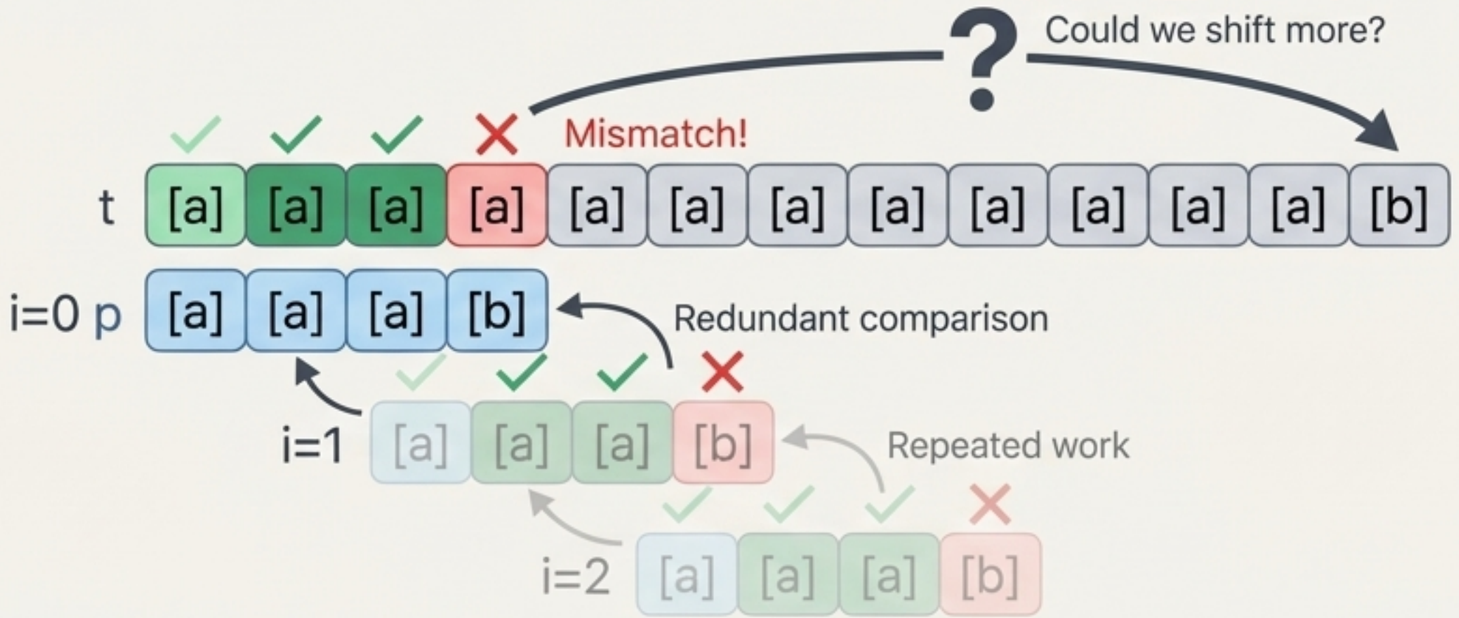
- The outer loop runs at most `n – m + 1` times.
  外层循环最多执行 `n – m + 1` 次。
- In the worst case, the inner loop runs `m` times for *each* outer loop iteration.
  在最坏情况下，内层循环对每次外层循环迭代都执行 `m` 次。
- This results in a total worst-case time complexity of **O((n-m)m)**.
  这导致总的最坏时间复杂度为 **O((n-m)m)**。



The “Naive” Nature
“朴素”之名

The algorithm is called "naive" because it does not learn from information gathered during mismatches. After a failed attempt, it mechanically shifts the pattern by only one position, even when the mismatch information could justify a larger, more intelligent shift.

该算法之所以“朴素”，是因为它不会从不匹配中学习。在一次失败的尝试后，它只是机械地将模式串移动一位，即使前一次比较的信息本可以证明一次更长、更智能的移位是可行的。



Summary: The Naive Algorithm
总结：朴素算法

Concept: An intuitive “slide and compare” strategy that exhaustively checks every possible starting position.

概念：一种直观的“滑动并比较”策略，它会彻底检查每一个可能的起始位置。

Implementation: Simple to understand and code, relying on two nested loops.

实现：易于理解和编码，依赖于两个嵌套循环。

Performance: A worst-case time complexity of O((n-m)m), making it inefficient for certain text and pattern combinations.

性能：最坏时间复杂度为O((n-m)m)，使其在处理特定文本和模式组合时效率低下。