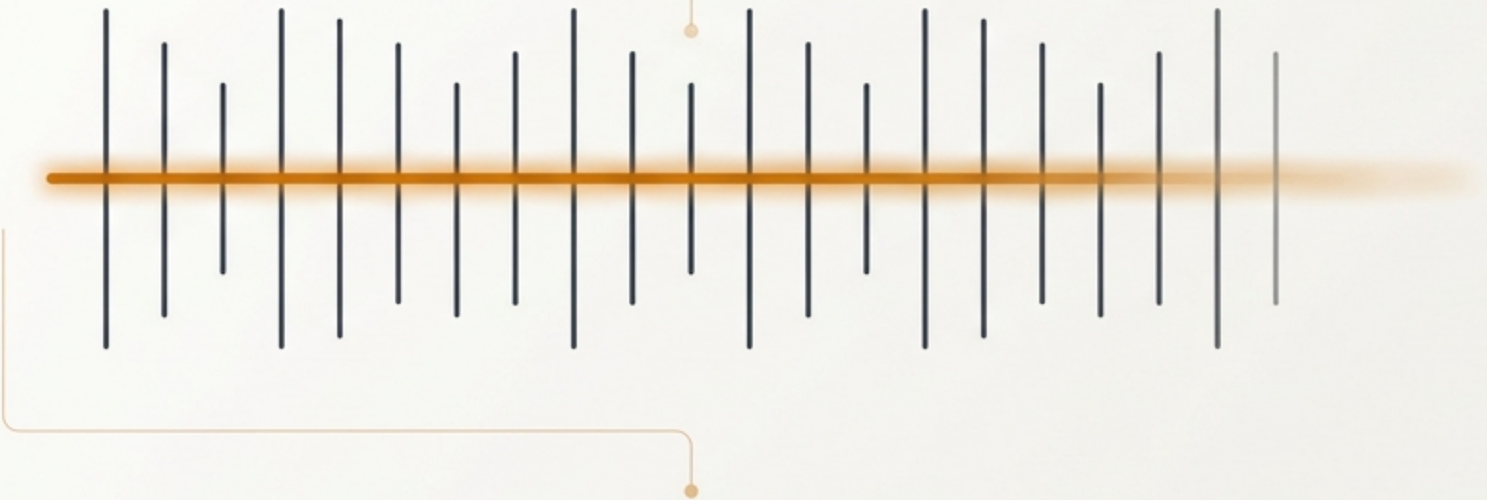
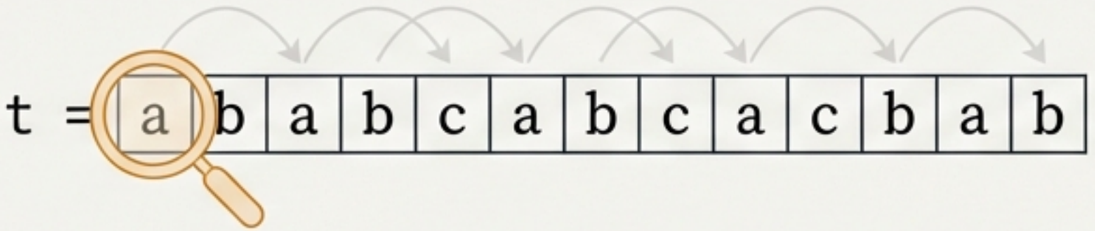
The Rabin-Karp Algorithm
Rabin-Karp 算法



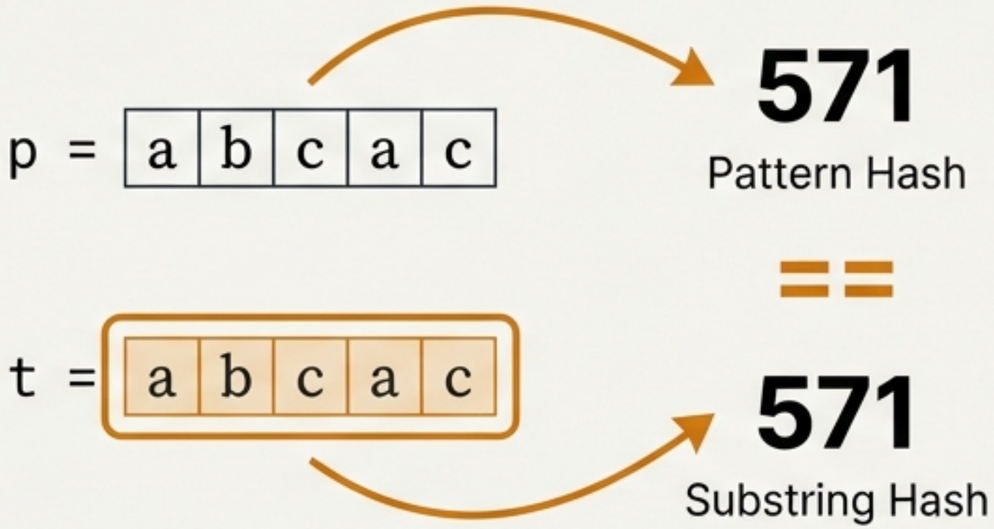
A Story of String Matching with Hashing
一个关于哈希字符串匹配的故事

Beyond Character-by-Character Comparison 超越逐字符比较

Slow / 逐字符



Fast Comparison / 快速比较



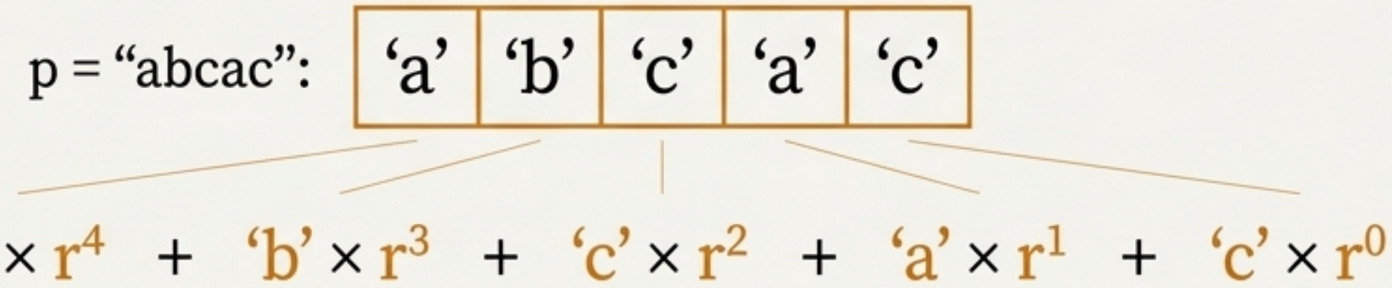
The core idea of the Rabin-Karp algorithm is to convert strings into numbers, or 'hash values.' Instead of comparing long strings of characters, we can quickly compare their **single-number fingerprints.** This provides a path to a much faster search.

Rabin-Karp 算法的核心思想是将字符串转换为数字，即“哈希值”。我们不再比较长字符串，而是先比较它们的单数字指纹。这为实现更快的搜索提供了可能。

Translating Strings to Numbers 将字符串转换为数字

We treat a string of length m as a number in a base-r system, where r is the size of the character set. The value x for a pattern p[i..i+m-1] is calculated as:
我们将长度为 m 的字符串视为一个 r 进制数（其中 r 是我们字符集的大小）。模式 p[i..i+m-1 的值 x 计算如下：

$$x_i = p[i]r^{m-1} + p[i+1]r^{m-2} + \ldots + p[i+m-1]r^0$$



To keep the numbers manageable, we perform all calculations modulo a large prime number q. The final hash value is $h(x_i) = x_i \bmod q$.
为了使数值可控，我们在一个大素数 q 的模下执行所有计算。最终哈希值为 $h(x_i) = x_i \bmod q$.

Initial Implementation: The `hash()` Function 初始实现：`hash()` 函数

```
long hash(const string& p, int i, int m) { // 计算子串的哈希值
    long h = 0;
    for(int j=0; j<m; j++)
        h = (r*h + p[i+j]) % q;
    return h;
}
```

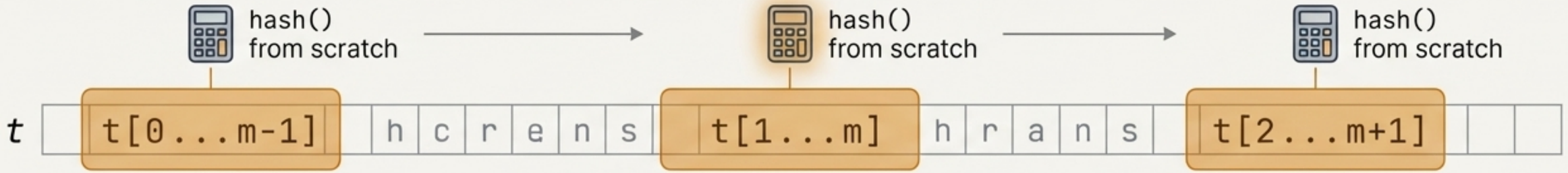
`long h = 0;`: Initializes the hash value.
（初始化哈希值）

`for(int j=0; j<m; j++)`: Iterates through each character of the substring.
（遍历子串中的每个字符）

`h = (r*h + p[i+j]) % q;`: This is the core calculation. It efficiently computes the polynomial hash using Horner's method, avoiding expensive exponentiation (r^m) inside the loop.
（这是核心计算。它使用霍纳法则高效地计算多项式哈希，避免了循环内昂贵的幂运算（`r^m`））

A First Attempt... And a Performance Bottleneck 首次尝试…及其性能瓶瓶颈

A simple approach is to calculate the pattern's hash once, then slide a window across the text, recalculating the hash for each substring from scratch and comparing.
一个简单的方法是先计算一次模式的哈希值，然后将窗口滑过文本，为每个子串从头计算哈希值并进行比较。



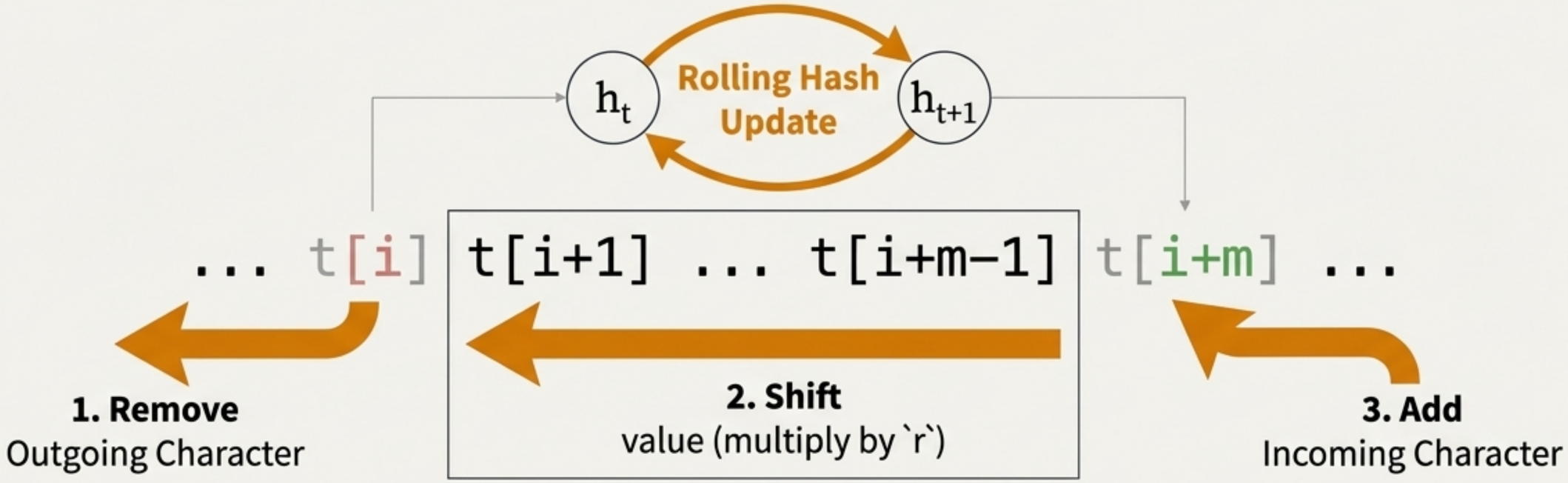

```
int Rabin_Karp(const string& t, const string& p) {
    int m = p.length();
    int n = t.length();
    long hp = hash(p, 0, m);
    for(int i=0; i <= n-m; i++) {
        long ht = hash(t, i, m); // The bottleneck!
        if(hp == ht)
            return i;
    }
    return n;
}
```

This is too slow. The hash() function takes O(m) time. We call it n-m+1 times, leading to a total time complexity of **O((n-m)m)**—no better than brute force.
这太慢了。hash() 函数耗时 O(m)。我们调用了 n-m+1 次，导致总时间复杂度为 **O((n-m)m)**——并不比暴力破解法好。

The Breakthrough: The Rolling Hash 突破口：滚动哈希

The solution is to avoid recalculation. We can compute the next hash value from the previous one in constant O(1) time.

解决方案是避免重新计算。我们可以用上一个哈希值在常数 O(1) 时间内计算出下一个哈希值。



$$x_{i+1} = (x_i - t[i] * r^{m-1}) * r + t[i+m]$$

Implementation: Preprocessing and Initialization
实现：预处理与初始化

Before we can start rolling the hash, we perform a one-time setup.
在开始滚动哈希之前，我们进行一次性设置。

```
int Rabin_Karp(const string& t, const string& p) {
    int m = p.length();
    int n = t.length();
    if(n < m) return n;

    long ht = hash(t, 0, m); // 1
    long hp = hash(p, 0, m); // 2

    long rm = 1; // 3
    for(int i=1; i < m; i++)
        rm = (r * rm) % q;

    if((hp == ht) && check(t, p, 0, m))
        return 0;
    // ... main loop follows
}
```

1. `ht`: Calculate the initial hash for the first `m` characters of the text.
(计算文本前 m 个字符的初始哈希值)

2. `hp`: Calculate the hash for the pattern we are searching for.
(计算我们搜索的模式的哈希值)

3. `rm`: Pre-calculate `r^(m-1) mod q`. This constant is needed to remove the most significant character during the rolling step.
(预计算 `r^(m-1) mod q`。这个常量用于在滚动步骤中移除最高位的字符)

Implementation: The Rolling and Checking Loop 实现：滚动与检查循环

Now, we slide the window across the text, efficiently updating the hash in O(1) time at each step.
现在，我们将窗口滑过文本，在每一步中以 O(1) 的时间高效地更新哈希值。

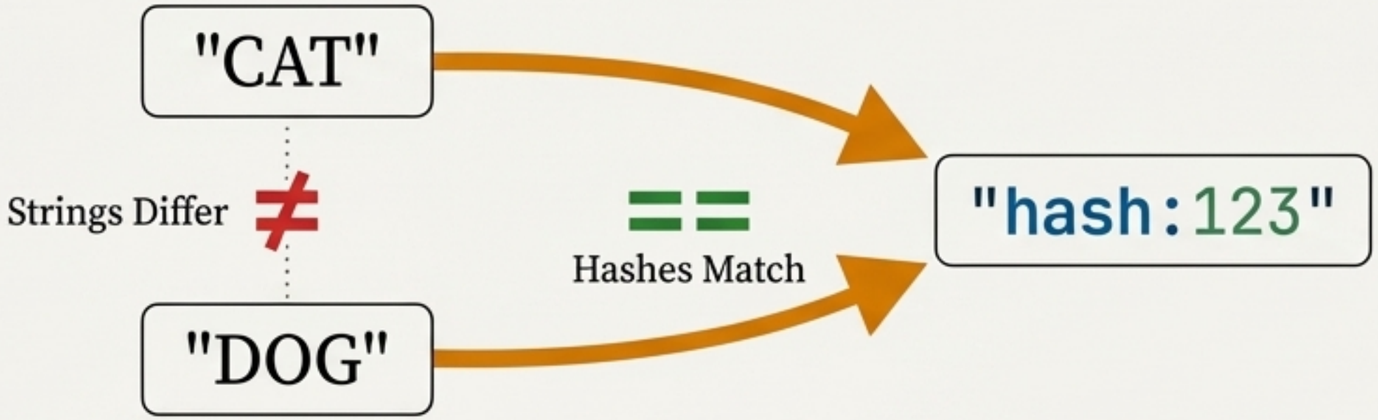
```
// ... initialization from previous slide
for(int i=m; i < n; i++) {
    // Remove leading char and add trailing char
    ht = (ht + q - rm * t[i-m] % q) % q;
    ht = (ht * r + t[i]) % q;

    int offset = i - m + 1;
    if((hp == ht) && check(t, p, offset, m)) {
        return offset;
    }
}
return n;  // Not found
```

The two highlighted lines efficiently update ht from the previous window's hash to the new one.
高亮的两行代码高效地将 ht 从上一个窗口的哈希值更新为新窗口的哈希值。

- Remove
- Add & Shift

The + q ensures the result of the subtraction remains positive in modular arithmetic. If the hashes match, check() performs a final character-by-character verification.
+ q 操作确保了在模运算中减法的结果保持为正。如果哈希值匹配，`check() 会执行最终的逐字符验证。

The Catch: Spurious Hits 注意事项：伪命中



It is possible for two different strings to have the same hash value. This is called a hash collision or a “spurious hit.”
两个不同的字符串可能会有相同的哈希值。这被称为哈希冲突或“伪命中”。

Therefore, `h(pattern) == h(substring)` does NOT guarantee `pattern == substring`.
因此，`h(模式) == h(子串)` 并不保证 `模式 == 子串`。

Solution: Whenever the hashes match, we must perform a full, character-by-character comparison to confirm it’s a true match. This is the purpose of the `check()` function, which is called after a hash match is found.
解决方案：每当哈希值匹配时，我们必须执行一次完整的逐字符比较，以确认是真正的匹配。这就是 `check()` 函数的目的，它在发现哈希匹配后被调用。

Algorithm Performance Analysis 算法性能分析

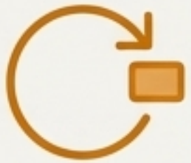
The algorithm's efficiency depends on the number of spurious hits. By choosing a large prime `q`, we make collisions extremely rare, ensuring excellent average-case performance. 该算法的效率取决于伪命中的数量。通过选择一个大的素数 `q`，我们可以使冲突变得极其罕见，从而确保优秀的平均情况性能。

Metric	Time Complexity	Notes
Preprocessing	O(m)	One-time calculation of initial hashes and `r^(m-1)`.
Worst-Case Match	O(nm)	A pathological case where every substring's hash collides with the pattern's hash.
Average-Case Match	**O(n+m)**	With a good hash function, collisions are rare. The search is dominated by the O(m) preprocessing and the single O(n) pass over the text.

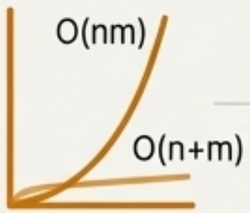
度量	时间复杂度	说明
预处理	O(m)	一次性计算初始哈希值和 r^(m-1)`。
最坏情况匹配	O(nm)	一种病态情况，每个子串的哈希值都与模式的哈希值冲突。
平均情况匹配	**O(n+m)**	使用好的哈希函数时，冲突很少。搜索时间主要由 O(m) 的预处理和对文本的单次 O(n) 遍历决定。

Rabin-Karp: The Complete Story Rabin-Karp 算法总结

Elegant Idea: Converts string comparison into a faster integer comparison using hashing.
优雅的思想：使用哈希将字符串比较转换为更快的整数比较。



Core Innovation: The O(1) “rolling hash” update avoids costly recalculations, making the algorithm highly efficient in practice.
核心创新：O(1) 的“滚动哈希”更新避免了昂贵的重复计算，使得该算法在实践中非常高效。



Practical Performance: Achieves an excellent O(n+m) average-case time, but requires careful implementation (a large prime `q`) to avoid the O(nm) worst case.
实际性能：实现了优秀的 O(n+m) 平均时间复杂度，但需要仔细实现（选择一个大的素数 `q`）以避免 O(nm) 的最坏情况。

Verification is Essential: Because of hash collisions (spurious hits), a character-level check on every hash match is mandatory to guarantee correctness.
验证至关重要：由于哈希冲突（伪命中），对每一次哈希匹配进行字符级检查是保证正确性的必要步骤。