KMP

这是一个经典的字符串匹配算法，该类问题一般为在一段字符串中匹配一个字串。我们可以第一时间想到的暴力解法就是一个一个的开始匹配，如果一旦有不匹配的则退出循环从主字符串的下一个字符继续匹配。时间复杂度在worst case是O(n*m)，n和m代表主串和子串的长度。

而KMP算法的主要思路是，当我们发现某个字符不匹配的时候，我们是否可以通过某种方法，来跳过遍历过的字符，从而避免我们之前提到的暴力算法的大量的重复匹配。那我们如何正确的跳过一些不必要的匹配呢?

这里我们需要引入一个概念，KMP的Next数组。让我们举个例子。假设我们要找的数组主串是abababca，我们需要匹配的子串是ababca。对应字串的next数组为next = [0, 0, 1, 2, 0, 1]。我们先不考虑怎么去构建这个next数组以及它的含义，我们先讨论怎么去使用它。

我们用两个指针，`i` 指向主串，`j` 指向子串。

1. 比较主串第 0 个字符 `a` 和子串第 0 个字符 `a`，相同 → 两个指针同时前进。
2. 主串第 1 个字符 `b` 和子串第 1 个字符 `b` 相同 → 前进。
3. 主串第 2 个字符 `a` 和子串第 2 个字符 `a` 相同 → 前进。
4. 主串第 3 个字符 `b` 和子串第 3 个字符 `b` 相同 → 前进。
5. 主串第 4 个字符 `a` 和子串第 4 个字符 `c` 不同 → 发生失配。

   这时我们不回到子串开头，而是利用 `next[3] = 2`，跳回子串第 2 个字符，继续尝试匹配。
6. 继续比较主串第 4 个字符 `a` 和子串第 2 个字符 `a`，匹配 → 指针继续前进。
7. 主串第 5 个字符 `b` 和子串第 3 个字符 `b`，匹配 → 前进。
8. 主串第 6 个字符 `c` 和子串第 4 个字符 `c`，匹配 → 前进。
9. 主串第 7 个字符 `a` 和子串第 5 个字符 `a`，匹配 → 前进。

这时子串已经匹配完成，说明我们在主串中找到了一个完整的匹配。

好的，我们已经讲解了如何使用next数组，那我们讲解一下如何正确构建next数组，还是使用上面的例子子串“ababca”。我们依次分析每一个元素的前后缀。

- 所谓“前缀”是指：从第一个字符开始，到某个字符为止的所有子串，但**不包含整个字符串本身**。
- “后缀”是指：从某个字符开始，一直到结尾的所有子串，但也**不包含整个字符串本身**。
- 我们在构造 `next[i]` 时，看的就是 `s[0:i+1]` 的前后缀中**最长的相同的那一段**。

第 0 位：`"a"`

- 前缀：空集
- 后缀：空集
- 公共部分长度：`0`

  `next[0] = 0`

第 1 位：`"ab"`

- 前缀：`["a"]`
- 后缀：`["b"]`
- 无交集 → 公共部分长度：`0`

  👉 `next[1] = 0`

第 2 位：`"aba"`

- 前缀：`["a", "ab"]`
- 后缀：`["ba", "a"]`
- 公共部分为：`"a"`，长度：`1`

  👉 `next[2] = 1`

第 3 位：`"abab"`

- 前缀：`["a", "ab", "aba"]`
- 后缀：`["bab", "ab", "b"]`
- 公共部分为：`"ab"`，长度：`2`

  👉 `next[3] = 2`

第 4 位：`"ababc"`

- 前缀：`["a", "ab", "aba", "abab"]`
- 后缀：`["babc", "abc", "bc", "c"]`
- 没有相同的前后缀 → 公共部分长度：`0`

  👉 `next[4] = 0`

第 5 位：`"ababca"`

- 前缀：`["a", "ab", "aba", "abab", "ababc"]`
- 后缀：`["babca", "abca", "bca", "ca", "a"]`
- 公共部分为 `"a"`，长度：`1`

  👉 `next[5] = 1`

所以最后的next数组为 = [0, 0, 1, 2, 0, 1]

```
def build_kmp_table(pattern):
    n = len(pattern)
    next = [0] * n  # Initialize the table with all zeros
    j = 0  # Length of the previous longest prefix-suffix

    for i in range(1, n):
        # If mismatch, backtrack to the previous prefix
        while j > 0 and pattern[i] != pattern[j]:
            j = next[j - 1]

        # If match, extend the current prefix
        if pattern[i] == pattern[j]:
            j += 1
            next[i] = j  # Update the table

    return next

def kmp_search(text, pattern):
    if not pattern:
        return 0  # Empty pattern is always found at index 0

    next = build_kmp_table(pattern)
```

```
    j = 0  # Pointer for pattern

    for i in range(len(text)):  # Pointer for text
        # If mismatch, follow the 'next' table to shift pattern
        while j > 0 and text[i] != pattern[j]:
            j = next[j - 1]

        # If match, move both pointers forward
        if text[i] == pattern[j]:
            j += 1

        # If we matched the entire pattern
        if j == len(pattern):
            return i - j + 1  # Match found at this index

    return -1  # No match found

# Example
text = "abababca"
pattern = "ababca"
result = kmp_search(text, pattern)
print("Pattern found at index:", result)
```

时间复杂度为O(n+m)