0028. 找出字符串中第一个匹配项的下标

ITCharge 大约 12 分钟

- 标签：双指针、字符串、字符串匹配
- 难度：中等

题目链接

- 0028. 找出字符串中第一个匹配项的下标 - 力扣

题目大意

描述：给定两个字符串 `haystack` 和 `needle` 。

要求：在 `haystack` 字符串中找出 `needle` 字符串出现的第一个位置（从 `0` 开始）。如果不存在，则返回 `-1` 。

说明：

- 当 `needle` 为空字符串时，返回 `0` 。
- $1 \leq haystack.length, needle.length \leq 10^4$。
- `haystack` 和 `needle` 仅由小写英文字符组成。

示例：

- 示例 1：

```py
输入：haystack = "hello", needle = "ll"
输出：2
解释："sad" 在下标 0 和 6 处匹配。第一个匹配项的下标是 0 ，所以返回 0 。
```

- 示例 2：

```py
输入：haystack = "leetcode", needle = "leeto"
输出：-1
解释："leeto" 没有在 "leetcode" 中出现，所以返回 -1 。
```

解题思路

字符串匹配的经典题目。常见的字符串匹配算法有：BF（Brute Force）算法、RK（Robin-Karp）算法、KMP（Knuth Morris Pratt）算法、BM（Boyer Moore）算法、Horspool 算法、Sunday 算法等。

思路 1：BF（Brute Force）算法

BF 算法思想：对于给定文本串 `T` 与模式串 `p`，从文本串的第一个字符开始与模式串 `p` 的第一个字符进行比较，如果相等，则继续逐个比较后续字符，否则从文本串 `T` 的第二个字符起重新和模式串 `p` 进行比较。依次类推，直到模式串 `p` 中每个字符依次与文本串 `T` 的一个连续子串相等，则模式匹配成功。否则模式匹配失败。

BF 算法具体步骤如下：

1. 对于给定的文本串 `T` 与模式串 `p`，求出文本串 `T` 的长度为 `n`，模式串 `p` 的长度为 `m`。
2. 同时遍历文本串 `T` 和模式串 `p`，先将 `T[0]` 与 `p[0]` 进行比较。
    1. 如果相等，则继续比较 `T[1]` 和 `p[1]`。以此类推，一直到模式串 `p` 的末尾 `p[m - 1]` 为止。
    2. 如果不相等，则将文本串 `T` 移动到上次匹配开始位置的下一个字符位置，模式串 `p` 则回退到开始位置，再依次进行比较。
3. 当遍历完文本串 `T` 或者模式串 `p` 的时候停止搜索。

思路 1：代码

```py
class Solution:
    def strStr(self, haystack: str, needle: str) -> int:
        i = 0
        j = 0
        len1 = len(haystack)
        len2 = len(needle)

        while i < len1 and j < len2:
            if haystack[i] == needle[j]:
                i += 1
```

```
                j += 1
            else:
                i = i - (j - 1)
                j = 0

        if j == len2:
            return i - j
        else:
            return -1
```

思路 1：复杂度分析

- **时间复杂度**：平均时间复杂度为 $O(n+m)$，最坏时间复杂度为 $O(m \times n)$。其中文本串 T 的长度为 n，模式串 p 的长度为 m。
- **空间复杂度**：$O(1)$。

思路 2：RK（Robin Karp）算法

RK 算法思想：对于给定文本串 `T` 与模式串 `p`，通过滚动哈希算快速筛选出与模式串 `p` 不匹配的文本位置，然后在其余位置继续检查匹配项。

RK 算法具体步骤如下：

1. 对于给定的文本串 `T` 与模式串 `p`，求出文本串 `T` 的长度为 `n`，模式串 `p` 的长度为 `m`。
2. 通过滚动哈希算法求出模式串 `p` 的哈希值 `hash_p`。
3. 再通过滚动哈希算法对文本串 `T` 中 `n - m + 1` 个子串分别求哈希值 `hash_t`。
4. 然后逐个与模式串的哈希值比较大小。
    1. 如果当前子串的哈希值 `hash_t` 与模式串的哈希值 `hash_p` 不同，则说明两者不匹配，则继续向后匹配。
    2. 如果当前子串的哈希值 `hash_t` 与模式串的哈希值 `hash_p` 相等，则验证当前子串和模式串的每个字符是否真的相等（避免哈希冲突）。
        1. 如果当前子串和模式串的每个字符相等，则说明当前子串和模式串匹配。
        2. 如果当前子串和模式串的每个字符不相等，则说明两者不匹配，继续向后匹配。
5. 比较到末尾，如果仍未成功匹配，则说明文本串 `T` 中不包含模式串 `p`，方法返回 `-1`。

思路 2：代码

```py
class Solution:
    def strStr(self, haystack: str, needle: str) -> int:
        def rabinKarp(T: str, p: str) -> int:
            len1, len2 = len(T), len(p)

            hash_p = hash(p)
            for i in range(len1 - len2 + 1):
                hash_T = hash(T[i: i + len2])
                if hash_p != hash_T:
                    continue
                k = 0
                for j in range(len2):
                    if T[i + j] != p[j]:
                        break
                    k += 1
                if k == len2:
                    return i
            return -1
        return rabinKarp(haystack, needle)
```

思路 1：复杂度分析

- **时间复杂度**：$O(n)$。其中文本串 T 的长度为 n，模式串 p 的长度为 m。
- **空间复杂度**：$O(m)$。

思路 3：KMP（Knuth Morris Pratt）算法

KMP 算法思想：对于给定文本串 `T` 与模式串 `p`，当发现文本串 `T` 的某个字符与模式串 `p` 不匹配的时候，可以利用匹配失败后的信息，尽量减少模式串与文本串的匹配次数，避免文本串位置的回退，以达到快速匹配的目的。

KMP 算法具体步骤如下：

1. 根据 `next` 数组的构造步骤生成「前缀表」 `next` 。

2. 使用两个指针 `i` 、 `j` ，其中 `i` 指向文本串中当前匹配的位置， `j` 指向模式串中当前匹配的位置。初始时， `i = 0` ， `j = 0` 。
3. 循环判断模式串前缀是否匹配成功，如果模式串前缀匹配不成功，将模式串进行回退，即 `j = next[j - 1]` ，直到 `j == 0` 时或前缀匹配成功时停止回退。
4. 如果当前模式串前缀匹配成功，则令模式串向右移动 `1` 位，即 `j += 1` 。
5. 如果当前模式串 **完全** 匹配成功，则返回模式串 `p` 在文本串 `T` 中的开始位置，即 `i - j + 1` 。
6. 如果还未完全匹配成功，则令文本串向右移动 `1` 位，即 `i += 1` ，然后继续匹配。
7. 如果直到文本串遍历完也未完全匹配成功，则说明匹配失败，返回 `-1` 。

思路 3：代码

py

```
class Solution:
    def strStr(self, haystack: str, needle: str) -> int:
        # KMP 匹配算法，T 为文本串，p 为模式串
        def kmp(T: str, p: str) -> int:
            n, m = len(T), len(p)

            next = generateNext(p)  # 生成 next 数组

            i, j = 0, 0
            while i < n and j < m:
                if j == -1 or T[i] == p[j]:
                    i += 1
                    j += 1
                else:
                    j = next[j]
            if j == m:
                return i - j

            return -1

        # 生成 next 数组
        # next[i] 表示坏字符在模式串中最后一次出现的位置
        def generateNext(p: str):
            m = len(p)

            next = [-1 for _ in range(m)]  # 初始化数组元素全部为 -1
            i, k = 0, -1
            while i < m - 1:  # 生成下一个 next 元素
                if k == -1 or p[i] == p[k]:
```

```
                i += 1
                k += 1
                if p[i] == p[k]:
                    next[i] = next[k]  # 设置 next 元素
                else:
                    next[i] = k  # 退到更短相同前缀
            else:
                k = next[k]
        return next

    return kmp(haystack, needle)
```

思路 3：复杂度分析

- **时间复杂度**：$O(n+m)$，其中文本串 T 的长度为 n，模式串 p 的长度为 m。
- **空间复杂度**：$O(m)$。

思路 4：BM（Boyer Moore）算法

BM 算法思想：对于给定文本串 `T` 与模式串 `p`，先对模式串 `p` 进行预处理。然后在匹配的过程中，当发现文本串 `T` 的某个字符与模式串 `p` 不匹配的时候，根据启发策略，能够直接尽可能地跳过一些无法匹配的情况，将模式串多向后滑动几位。

BM 算法具体步骤如下：

1. 计算出文本串 `T` 的长度为 `n`，模式串 `p` 的长度为 `m`。
2. 先对模式串 `p` 进行预处理，生成坏字符位置表 `bc_table` 和好后缀规则后移位数表 `gs_talbe`。
3. 将模式串 `p` 的头部与文本串 `T` 对齐，将 `i` 指向文本串开始位置，即 `i = 0`。`j` 指向模式串末尾位置，即 `j = m - 1`，然后从模式串末尾位置开始进行逐位比较。
    1. 如果文本串对应位置 `T[i + j]` 上的字符与 `p[j]` 相同，则继续比较前一位字符。
        1. 如果模式串全部匹配完毕，则返回模式串 `p` 在文本串中的开始位置 `i`。
    2. 如果文本串对应位置 `T[i + j]` 上的字符与 `p[j]` 不相同，则：
        1. 根据坏字符位置表计算出在「坏字符规则」下的移动距离 `bad_move`。
        2. 根据好后缀规则后移位数表计算出在「好后缀规则」下的移动距离 `good_mode`。
        3. 取两种移动距离的最大值，然后对模式串进行移动，即 `i += max(bad_move, good_move)`。
4. 如果移动到末尾也没有找到匹配情况，则返回 `-1`。

思路 4：代码

```py
class Solution:
    def strStr(self, haystack: str, needle: str) -> int:
        def boyerMoore(T: str, p: str) -> int:
            n, m = len(T), len(p)

            bc_table = generateBadCharTable(p)      # 生成坏字符位置表
            gs_list = generageGoodSuffixList(p)     # 生成好后缀规则后移位数表

            i = 0
            while i <= n - m:
                j = m - 1
                while j > -1 and T[i + j] == p[j]:
                    j -= 1
                if j < 0:
                    return i
                bad_move = j - bc_table.get(T[i + j], -1)
                good_move = gs_l[j]
                i += max(bad_mov    ood_move)
            return -1

        # 生成坏字符位置表
        def generateBadCharTable(p: str):
            bc_table = dict()

            for i in range(len(p)):
                bc_table[p[i]] = i          # 坏字符在模式串中最后一次出现的位置
            return bc_table

        # 生成好后缀规则后移位数表
        def generageGoodSuffixList(p: str):
            m = len(p)
            gs_list = [m for _ in range(m)]
            suffix = generageSuffixArray(p)
            j = 0
            for i in range(m - 1, -1, -1):
                if suffix[i] == i + 1:
                    while j < m - 1 - i:
                        if gs_list[j] == m:
```

```
                        gs_list[j] = m - 1 - i
                    j += 1

        for i in range(m - 1):
            gs_list[m - 1 - suffix[i]] = m - 1 - i

        return gs_list

    def generageSuffixArray(p: str):
        m = len(p)
        suffix = [m for _ in range(m)]
        for i in range(m - 2, -1, -1):
            start = i
            while start >= 0 and p[start] == p[m - 1 - i + start]:
                start -= 1
            suffix[i] = i - start
        return suffix

    return boyerMoore(haystack, needle)
```

思路 4：复杂度分析

- **时间复杂度**：$O(n+\sigma)$，其中文本 T 的长度为 n，字符集的大小是 σ。
- **空间复杂度**：$O(m)$。其中模式串 p 的长度为 m。

思路 5：Horspool 算法

Horspool 算法思想：对于给定文本串 `T` 与模式串 `p`，先对模式串 `p` 进行预处理。然后在匹配的过程中，当发现文本串 `T` 的某个字符与模式串 `p` 不匹配的时候，根据启发策略，能够尽可能的跳过一些无法匹配的情况，将模式串多向后滑动几位。

Horspool 算法具体步骤如下：

1. 计算出文本串 `T` 的长度为 `n`，模式串 `p` 的长度为 `m`。
2. 先对模式串 `p` 进行预处理，生成后移位数表 `bc_table`。
3. 将模式串 `p` 的头部与文本串 `T` 对齐，将 `i` 指向文本串开始位置，即 `i = 0`。`j` 指向模式串末尾位置，即 `j = m - 1`，然后从模式串末尾位置开始比较。
    1. 如果文本串对应位置的字符 `T[i + j]` 与模式串对应字符 `p[j]` 相同，则继续比较前一位字符。
        1. 如果模式串全部匹配完毕，则返回模式串 `p` 在文本串中的开始位置 `i`。
    2. 如果文本串对应位置的字符 `T[i + j]` 与模式串对应字符 `p[j]` 不同，则：

    1. 根据后移位数表 `bc_table` 和模式串末尾位置对应的文本串上的字符 `T[i + m - 1]` ，计算出可移动距离 `bc_table[T[i + m - 1]]` ，然后将模式串进行后移。
4. 如果移动到末尾也没有找到匹配情况，则返回 `-1` 。

思路 5：代码

```py
class Solution:
    def strStr(self, haystack: str, needle: str) -> int:
        def horspool(T: str, p: str) -> int:
            n, m = len(T), len(p)

            bc_table = generateBadCharTable(p)

            i = 0
            while i <= n - m:
                j = m - 1
                while j > -1 and T[i + j] == p[j]:
                    j -= 1
                if j < 0:
                    return i
                i += bc_table.ge+/'T'[i + m - 1], m)
            return -1

        # 生成后移位置表
        # bc_table[bad_char] 表示坏字符在模式串中最后一次出现的位置
        def generateBadCharTable(p: str):
            m = len(p)
            bc_table = dict()

            for i in range(m - 1):
                bc_table[p[i]] = m - i - 1    # 更新坏字符在模式串中最后一次出现的
位置
            return bc_table

        return horspool(haystack, needle)
```

思路 5：复杂度分析

- **时间复杂度**：$O(n)$。其中文本串 T 的长度为 n。
- **空间复杂度**：$O(m)$。其中模式串 p 的长度为 m。

思路 6：Sunday 算法

Sunday 算法思想：对于给定文本串 `T` 与模式串 `p`，先对模式串 `p` 进行预处理。然后在匹配的过程中，当发现文本串 `T` 的某个字符与模式串 `p` 不匹配的时候，根据启发策略，能够尽可能的跳过一些无法匹配的情况，将模式串多向后滑动几位。

Sunday 算法具体步骤如下：

1. 计算出文本串 `T` 的长度为 `n`，模式串 `p` 的长度为 `m`。
2. 先对模式串 `p` 进行预处理，生成后移位数表 `bc_table`。
3. 将模式串 `p` 的头部与文本串 `T` 对齐，将 `i` 指向文本串开始位置，即 `i = 0`。`j` 指向模式串末尾位置，即 `j = m - 1`，然后从模式串末尾位置开始比较。
    1. 如果文本串对应位置的字符 `T[i + j]` 与模式串对应字符 `p[j]` 相同，则继续比较前一位字符。
        1. 如果模式串全部匹配完毕，则返回模式串 `p` 在文本串中的开始位置 `i`。
    2. 如果文本串对应位置的字符 `T[i + j]` 与模式串对应字符 `p[j]` 不同，则：
        1. 根据后移位数表 `bc_table` 和模式串末尾位置对应的文本串上的字符 `T[i + m - 1]`，计算出可移动距离 `bc_table[T[i + m - 1]]`，然后将模式串进行后移。
4. 如果移动到末尾也没有找到匹配情况，则返回 `-1`。

思路 6：代码

py

```
class Solution:
    def strStr(self, haystack: str, needle: str) -> int:
        # sunday 算法，T 为文本串，p 为模式串
        def sunday(T: str, p: str) -> int:
            n, m = len(T), len(p)
            if m == 0:
                return 0

            bc_table = generateBadCharTable(p)          # 生成后移位数表

            i = 0
            while i <= n - m:
                if T[i: i + m] == p:
                    return i                            # 匹配完成，返回模式串 p 在文本串 T 中的位置
                if i + m >= n:
```

```
                    return -1
                i += bc_table.get(T[i + m], m + 1)       # 通过后移位数表，向右进行
进行快速移动
            return -1                                    # 匹配失败

        # 生成后移位数表
        # bc_table[bad_char] 表示遇到坏字符可以向右移动的距离
        def generateBadCharTable(p: str):
            m = len(p)
            bc_table = dict()

            for i in range(m):
                bc_table[p[i]] = m - i                   # 更新遇到坏字符可向右移动的距
离
            return bc_table

        return sunday(haystack, needle)
```

思路 6：复杂度分析

- **时间复杂度**：$O(n)$。其中文本串 T 的长度为 n。
- **空间复杂度**：$O(m)$。其中模式串 P 长度为 m。