13 实现 strStr() 函数

更新时间：2019-08-21 09:48:02

> 知识犹如人体的血液一样宝贵。
>
> ——高士其

刷题内容

> **难度: Easy**
>
> 原题链接：https://leetcode-cn.com/problems/implement-strstr/。

内容描述

实现 strStr() 函数。

给定一个 haystack 字符串和一个 needle 字符串，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。

示例 1:
输入: haystack = "hello", needle = "ll"
输出: 2

示例 2:
输入: haystack = "aaaaa", needle = "bba"
输出: -1

说明: 当 needle 是空字符串时，我们应当返回什么值呢？这是一个在面试中很好的问题。

对于本题而言，当 needle 是空字符串时我们应当返回 0 。这与C语言的 strstr() 以及 Java的 indexOf() 定义相符。

题目详解

在做这道题之前，要考虑以下几种情况：

- haystack 为空字符串呢？
- needle 为空字符串呢？
- needle 长度比 haystack 更长呢？
- 等等

解题方案

思路 1：时间复杂度: O(m * n) 空间复杂度: O(1)

这道题我们可以直接用 Python 自带的 api `str.find()`来解答。这个函数输入一个参数 `a`，然后在str中找到第一次出现子串 `a` 的索引，如果不存在就返回 `-1`。

> python 的str.find() 底层实现是 Boyer–Moore–Horspool 算法，这个算法的平均时间复杂度是O(N)，而最差的时间复杂度为O(NM)。
>
> 时间复杂度解释：
>
> - m = len(haystack)
> - n = len(needle)

这里给出参考：Runtime of python's if substring in string。

Python beats 100%

```
class Solution:
    def strStr(self, haystack: str, needle: str) -> int:
        return haystack.find(needle)
```

c++ beats 93.94 %

c++自带的api也是 `find`，总感觉使用这个函数有点投机取巧的意思：

```
class Solution {
public:
    int strStr(string haystack, string needle) {
        return haystack.find(needle);
    }
};
```

java beats 100%

java自带的定位子串位置的api是 `indexOf`：

```
class Solution {
    public int strStr(String haystack, String needle) {
        return haystack.indexOf(needle);
    }
}
```

go beats 100%

go自带的定位字串位置的api是 `strings.Index`：

```
func strStr(haystack string, needle string) int {
    return strings.Index(haystack, needle)
}
```

思路 2：时间复杂度：O(m + n) 空间复杂度：O(n)

看到关于字符串匹配类型的问题，这里我要给著名的 KMP 算法 打 call。如果不了解 KMP 算法的同学可以先看一下从头到尾彻底理解KMP（2014年8月22日版） 这篇文章。

> 在计算机科学中，**Knuth-Morris-Pratt字符串查找算法**（简称为**KMP算法**）可在一个**主文本字符串** `S` 内查找一个**词** `W` 的出现位置。此算法通过运用对这个词在不匹配时本身就包含足够的信息来确定下一个匹配将在哪里开始的发现，从而避免重新检查先前匹配的字符。
>
> 这个算法是由高德纳和沃恩·普拉特在1974年构思，同年詹姆斯·H·莫里斯也独立地设计出该算法，最终由三人于1977年联合发表。
>
> KMP的时间复杂度为O(m+n)
>
> —来自维基百科

Python beats 25.41%

你可能会问，使用 KMP 算法还没有使用 `find()` 快，我干嘛还要用这个算法呢？其实这是和测试的字符串有关系。测试的字符串越长，则 KMP 算法相较于 `find()` 的优势才更明显。不过使用两种方法还是要看具体情况。我一直说在最合适的情况下使用最合适的算法才是最佳。

```python
class Solution:
    def strStr(self, haystack: str, needle: str) -> int:
        text, pattern = haystack, needle
        if not pattern:
            return 0

        lps = self.findLPS(pattern) # longest proper prefix which is also suffix
        i, j = 0, 0 # idx for text and pattern
        res = -1
        while i < len(text):
            if pattern[j] == text[i]:
                i += 1
                j += 1
            if j == len(pattern):
                res = i - j
                return res
            elif i < len(text) and pattern[j] != text[i]: # mismatch after j matches
                if j != 0: # Don't match lps[0..lps[j-1]] characters, they will match anyway
                    j = lps[j-1]
                else:
                    i += 1
        return -1

    def findLPS(self, pattern):
        length, lps = 0, [0]
        for i in range(1, len(pattern)):
            while length > 0 and pattern[length] != pattern[i]:
                length = lps[length-1]
            if pattern[length] == pattern[i]:
                length += 1
            lps.append(length)
        return lps
```

c++ beats 54.32%*

```cpp
class Solution {
public:
    vector<int> callLps(string pattern) {
        vector<int> lps(pattern.size(), 0);
        int length = 0;
        for (int i = 1;i < pattern.size();i++) {
            while (length > 0 && pattern[length] != pattern[i]) {
                length = lps[length - 1];
            }
            if (pattern[length] == pattern[i]) {
                length++;
            }
            lps[i] = length;
        }
        return lps;
    }
    int strStr(string haystack, string needle) {
        if (needle.size() == 0) {
            return 0;
        }
        vector<int> lps = callLps(needle); // longest proper prefix which is also suffix
        int i = 0, j = 0; // idx for text and pattern
        while (i < haystack.size()) {
            if (needle[j] == haystack[i]) {
                i++;
                j++;
            }
            if (j == needle.size()) {
                return i - j;
            } else if (i < haystack.size() && needle[j] != haystack[i]) { // mismatch after j matches
                if (j != 0) { // Don't match lps[0..lps[j-1]] characters, they will match anyway
                    j = lps[j - 1];
                } else {
                    i++;
                }
            }
        }
        return -1;
    }
};
```

java beats 13.52 %

```java
class Solution {
    private int[] callLps(String pattern) {
        int[] lps = new int[pattern.length()];
        lps[0] = 0;
        int length = 0;
        for (int i = 1;i < pattern.length();i++) {
            while (length > 0 && pattern.charAt(length) != pattern.charAt(i)) {
                length = lps[length - 1];
            }
            if (pattern.charAt(length) == pattern.charAt(i)) {
                length++;
            }
            lps[i] = length;
        }
        return lps;
    }
    public int strStr(String haystack, String needle) {
        if ("".equals(needle)) {
            return 0;
        }
        int[] lps = callLps(needle); // longest proper prefix which is also suffix
        int i = 0, j = 0; // idx for text and pattern
        while (i < haystack.length()) {
            if (needle.charAt(j) == haystack.charAt(i)) {
                i++;
                j++;
            }
            if (j == needle.length()) {
                return i - j;
            } else if (i < haystack.length() && needle.charAt(j) != haystack.charAt(i)) { // mismatch after j matches
                if (j > 0) { // Don't match lps[0..lps[j-1]] characters, they will match anyway
                    j = lps[j - 1];
                } else {
                    i++;
                }
            }
        }
        return -1;
    }
}
```

go beats 100%

```go
func callLps(pattern string) []int {
    lps := make([]int, len(pattern))
    lps[0] = 0
    length := 0
    for i := 1; i < len(pattern); i++ {
        for length > 0 && pattern[length] != pattern[i] {
            length = lps[length - 1]
        }
        if pattern[length] == pattern[i] {
            length++
        }
        lps[i] = length
    }
    return lps
}

func strStr(haystack string, needle string) int {
    if needle == "" {
        return 0
    }
    lps := callLps(needle) // longest proper prefix which is also suffix
    i, j := 0, 0 // idx for text and pattern
    for i < len(haystack) {
        if needle[j] == haystack[i] {
            i++
            j++
        }
        if j == len(needle) {
            return i - j
        } else if i < len(haystack) && needle[j] != haystack[i] { // mismatch after j matches
            if j > 0 { // Don't match lps[0..lps[j-1]] characters, they will match anyway
                j = lps[j - 1]
            } else {
                i++
            }
        }
    }
    return -1
}
```

小结

- 用一门熟悉的语言去刷题，一定要把比较爽的一些api记下来，最好还要知道底层的逻辑。这样可以帮助我们节省时间，对算法复杂度的理解也更加游刃有余；
- 可以像我一样，学习了一个算法就自己写一个小文档，然后记录下来，下次碰到可以用这个算法的时候就直接贴自己的模版，贴模版解题要方便得多。最好再加几个小例子帮助自己快速加深记忆，以后面试的时候捡起来也比较快。

}

12 移除数组中的指定元素

14 两数相加