151. 反转字符串中的单词

题目描述

给你一个字符串 s，请你反转字符串中**单词**的顺序。

- **单词**是由非空格字符组成的字符串。
- s 中使用至少一个空格将字符串中的**单词**分隔开。

返回**单词顺序颠倒**且**单词之间用单个空格连接**的结果字符串。

注意：

- 输入字符串 s 中可能会存在前导空格、尾随空格或者单词间的多个空格。
- 返回的结果字符串中，单词间应当仅用单个空格分隔，且不包含任何额外的空格。

示例

示例 1：

```
输入：s = "the sky is blue"
输出："blue is sky the"
```

示例 2：

```
输入：s = "  hello world  "
输出："world hello"
解释：反转后的字符串中不能存在前导空格和尾随空格。
```

示例 3：

```
输入：s = "a good   example"
输出："example good a"
解释：如果两个单词间有多余的空格，反转后的字符串需要将单词间的空格减少到仅有一个。
```

提示

- $$1 \leq s.length \leq 10^4$$
- s 包含英文大小写字母、数字和空格 `' '`
- s 中**至少存在一个单词**

进阶

- 如果字符串在你使用的编程语言中是一种可变数据类型，请尝试使用 $O(1)$ 额外空间复杂度的**原地解法**。

In [ ]:
```python
# python库，但是会使用额外空间
class Solution:
    def reverseWords(self, s: str) -> str:
        return " ".join(s.split()[::-1])
```

In [ ]:
```python
# 实际是三个双指针

class Solution:
    def trim_spaces(self , s):
        # 双指针
        left, right = 0, len(s) - 1
        while left<=right and s[left] == ' ':
            left += 1
        while left<=right and s[right] ==' ':
            right = right-1 # 用改变指针的方法去掉空格?

        tmp = []
        # 处理单词中间的空格
        while left<=right:
            if s[left] != " ":
                tmp.append(s[left])
            elif not tmp or tmp[-1] != ' ': # 这里加入了一个判断，如果 tmp 是空列表，tmp[-1] 会抛出 IndexError，  所以确保不为空
                tmp.append(s[left])
            left += 1
        return tmp

    def reverse_string(self, list, left, right):
        while left<right:
            # 从list的头和尾反转
            list[left], list[right] = list[right], list[left]
            left += 1
            right -= 1
        return None
```

```
    def reverse_each_word(self, word):
        left, right = 0, 0
        word_len = len(word)
        while left < word_len:
            while right < word_len and word[right] != ' ':
                right += 1
            self.reverse_string(word, left, right - 1)
            left = right + 1
            right = left  # 重置 `right` 与 `left` 同步


    def reverseWords(self, s: str) -> str:
        letter_list = self.trim_spaces(s)
        self.reverse_string(letter_list, 0, len(letter_list) -1)
        self.reverse_each_word(letter_list)
        return ''.join(letter_list)
```

55. 右旋字符串

题目描述

字符串的右旋转操作是将字符串尾部的若干个字符转移到字符串的前面。给定一个字符串 s 和一个正整数 k，请编写一个函数，将字符串中的后面 k 个字符移到字符串的前面，实现字符串的右旋转操作。

例如，对于输入字符串 `"abcdefg"` 和整数 $k = 2$，函数应该将其转换为 `"fgabcde"`。

输入描述

输入共包含两行：

1. 第一行为一个正整数 k，代表右旋转的位数。
2. 第二行为字符串 s，代表需要旋转的字符串。

输出描述

输出共一行，为进行了右旋转操作后的字符串。

示例

输入：

```
2
abcdefg
```

In [8]:
```
class Solution:
    # 第一次尝试，用string split，通过
    # python中字符串是不可变的，所以也需要额外空间
    def rotateString_split(self, k: int, s: str) -> str:
        str_len = len(s)
        string_1 = s[str_len-k:]
        string_2 = s[:str_len-k]
        new_string = string_1 + string_2
        print(new_string)
        return new_string

    # 答案中的简洁写法
    def rotateString_clean(self, k: int, s: str) -> str:
        s = s[len(s)-k:] + s[:len(s)-k]
        return s

    # 用python列表模拟答案的调换两次写法
    def rotateString(self, k: int, s: str) -> str:
        """
        使用三次反转法实现字符串右旋
        :param k: 右旋的位数
        :param s: 输入的字符串
        :return: 右旋后的字符串
        """
        # 确保旋转位数不超过字符串长度
        k = k % len(s)  # 右旋 k 等价于右旋 k % length

        # 将字符串转换为列表，因为字符串是不可变的
        s_list = list(s)

        # 1. 整体反转
        s_list.reverse()
        # 示例: "abcdefg" -> "gfedcba"

        # 2. 反转前 k 个字符
        s_list[:k] = reversed(s_list[:k])
        # 示例: 前 k=2 个字符反转后，"gfedcba" -> "fgedcba"
```

```
        # 3. 反转后 length-k 个字符
        s_list[k:] = reversed(s_list[k:])
        # 示例: 后 length-k=5 个字符反转后, "fgedcba" -> "fgabcde"

        # 将列表转换回字符串并返回
        return ''.join(s_list)


def test():
    solution = Solution()

    # 示例测试用例
    assert solution.rotateString(2, "abcdefg") == "fgabcde"
    assert solution.rotateString(3, "abcdefg") == "efgabcd"

    # 边界测试
    assert solution.rotateString(0, "abcdefg") == "abcdefg"  # k=0, 字符串保持不变
    assert solution.rotateString(7, "abcdefg") == "abcdefg"  # k 等于字符串长度
    assert solution.rotateString(8, "abcdefg") == "gabcdef"  # k 超过字符串长度, 等价于 k=1
    assert solution.rotateString(1, "a") == "a"  # 单字符测试

    # 大数据测试
    assert solution.rotateString(3, "a" * 10000) == "aaa" * 3333 + "a"

    print("所有测试用例通过！")

# 调用测试函数
test()
```

所有测试用例通过！

知识点：

- python中list的反转用的是 reversed(list), 需要重新赋值

  `s_list[:k] = reversed(s_list[:k])`
- 三次反转法 "* **整体反转**： * 使用 `s_list.reverse()` 将整个字符串反转。
- 反转后的效果：原字符串 `"abcdefg"` -> `"gfedcba"`。
- **反转前 n 个字符**： * 使用切片 `s_list[:n]` 对前 n 个字符进行反转。
- 示例： `n=2` 时，反转 `"gfedcba"` 的前两位得到 `"fgedcba"`。
- **反转后 length-n 个字符**： * 使用切片 `s_list[n:]` 对后 length-n 个字符进行反转。
- 示例：后 5 个字符反转后，`"fgedcba"` -> `"fgabcde"`。"
- 在 C++ 中，字符串是可变的，可以原地修改，因此三次反转法可以做到 $O(1)$ 空间复杂度。
- 在 Python 中，由于字符串是不可变的，需要借助列表来模拟修改，因此存在 $O(n)$ 的空间开销。

28. 找出字符串中第一个匹配项的下标

题目描述

给你两个字符串 $haystack$ 和 $needle$，请你在 $haystack$ 字符串中找出 $needle$ 字符串的第一个匹配项的下标（下标从 0 开始）。如果 $needle$ 不是 $haystack$ 的一部分，则返回 -1。

示例

示例 1:

```
输入: haystack = "sadbutsad", needle = "sad"
输出: 0
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:

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

当 needle 是空字符串时我们应当返回 0

In [9]:

知识点： KMP (Knuth, Morris和Pratt) 算法 KMP主要应用在字符串匹配上。

KMP的主要思想是当出现字符串不匹配时，可以知道一部分之前已经匹配的文本内容，可以利用这些信息避免从头再去做匹配了。

next数组就是一个前缀表 (prefix table)。

前缀表是用来回退的，它记录了模式串与主串(文本串)不匹配的时候，模式串应该从哪里开始重新匹配。

要在文本串：aabaabaafa 中查找是否出现过一个模式串：aabaaf。

```
前缀表：记录下标i之前（包括i）的字符串中，有多大长度的相同前缀后缀。

后缀：  只包含尾字母不包含首字母的所有字串 f，af，aaf，baaf，abaaf，

aabaaf不是后缀因为a是首字母

前缀：包含首字母不包含尾字母的所有字串
a，aa，aab，aaba，aabaa，aabaaf


最长相等前后缀：
a：0
aa：1 前缀a = 后缀a
aab：  0
aaba：  长度1，a和a
aabaa：长度2，a和a，aa和aa
aabaaf：0


前缀表就是

[0,1,0,1,2,0]
对应
[a,a,b,a,a,f]


通过查找前缀表
例如遇到aabaaf，f时，查找aabaa，发现前缀表对应2，2代表最长相等前缀是aa，我们可以从aabaa的b中开始，也就是index对应2，就是查表得到的2，因为index从0开始


next()函数就在做这个功能，查前缀表，输出下一个
```

```
  Cell In[9], line 1
    知识点：  KMP（Knuth，Morris和Pratt）算法 KMP主要应用在字符串匹配上。
          ^
SyntaxError: invalid character '：' (U+FF1A)
```

In [ ]:

459. 重复的子字符串 简单 相关标签 相关企业 给定一个非空的字符串 s ，检查是否可以通过由它的一个子串重复多次构成。

示例 1:

输入: s = "abab" 输出: true 解释: 可由子串 "ab" 重复两次构成。 示例 2:

输入: s = "aba" 输出: false 示例 3:

输入: s = "abcabcabcabc" 输出: true 解释: 可由子串 "abc" 重复四次构成。 (或子串 "abcabc" 重复两次构成。)

提示：

1 <= s.length <= 104 s 由小写英文字母组成

这两题太难，无法集中注意力， 先跳过