“小萌，小帅，今天给你们留个作业：如果给你们两个字符串 A 和 B，判断 B 是否是 A 的子串，并返回 B 在 A 中第一次出现的位置，你们要如何实现？明天课上告诉我答案啊。”

“OK，No problem！老师，明天见！”

一夜无话，又到了上课的时间，小萌小帅早已来到了教室，脸上洋溢着自信的微笑……

“小萌，小帅，昨天的问题你们想到答案了吗？”

“哈哈！我和小萌都想到答案了！只要将字符串 A 和 B 对齐，一个一个字符比较，不匹配就让 B 和 A 的第二个字符对齐，依次比较……”

“还不错，你们的这个算法有个名字，叫作暴力匹配算法，今天我们就来讲一讲字符串匹配算法吧。”

5.1 字符串暴力匹配算法（BF 算法）

1. 算法介绍

字符串暴力匹配算法又称 BF（Brute Force）算法，它是一种最符合人类

思维习惯的算法。其算法原理简单暴力，因此得名。具体做法如下。

“为了统一概念，在后文中，我们把字符串 A 称为主串，把字符串 B 称为模式串。”

首先将主串和模式串左端对齐，逐一比较；如果第一个字符不能匹配，则模式串向后移动一位继续比较；如果第一个字符匹配，则继续比较后续字符；当某个字符发生不匹配时，模式串后移一位，如此往复，直至全部匹配或模式串移动到超过主串尾部，匹配失败。

图 5-1 展示了算法的具体匹配过程。

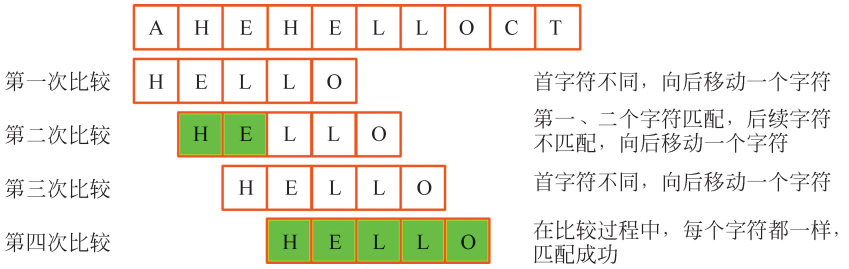

图 5-1　暴力匹配算法演示

2. 代码实现

原理比较简单，下面我们就使用代码来实现它。字符串暴力匹配算法实现如下。

```
#text 为主串，pattern 为模式串
def bf(text, pattern):
    i , j = 0 , 0
    while i < len(text) and j < len(pattern):
        # 如果字符匹配，则继续比较后续字符
        if(text[i] ==  pattern[j]):
            i += 1
            j += 1
        # 如果不匹配，模式串索引回溯到起始位置
        # 主串回溯位置到比较开始时的下一个字符
        else:
```

```
            i = i - j + 1
            j = 0
    # 如果模式串的所有字符都匹配成功，返回第一个匹配字符的位置
    if(j >= len(pattern)):
        return i - len(pattern)
    # 匹配失败，返回 -1
    else:
        return -1
```

【问题 5-1】 使用 BF 算法实现如下字符串匹配程序：给定一个主串“abcokabkoh”和一个模式串“abk”，找出模式串在主串中第一次出现的位置，若不存在则返回 –1。

3. 算法复杂度

暴力匹配算法的时间复杂度和空间复杂度按照以下方法进行计算。

假设主串长度为 n，模式串长度为 m。从算法原理可知，主串中的每个字符都可能与模式串中的字符进行一一比较，因此最坏情况下的比较次数为 $n \times m$，故算法的时间复杂度为 $O(nm)$。当 n 和 m 值较小时，算法效率尚可，但当 m 和 n 值较大时，算法效率下降明显。

暴力匹配算法不需要使用额外的存储空间，其空间复杂度为 $O(1)$。

5.2 字符串匹配 KMP 算法

1. 算法介绍

在暴力匹配算法中，每当发生字符不匹配时，根据算法策略，模式串后移

一位开始新一轮的比较，这也正是暴力匹配算法效率较低的原因。如果我们能在发生不匹配时打破后移一位的限制，动态确定模式串后移的长度，移动到下一个最可能发生匹配的位置，那么算法的效率将大大提升，可这样的算法真的存在吗？当然有，那就是著名的 KMP 匹配算法。

KMP 算法使用三位发明者 Knuth、Morris 和 Pratt 名字的首字母命名，是字符串匹配最经典的算法之一。

KMP 算法的原理较为复杂，我们就先通过案例来理解吧！使用 KMP 算法对如图 5-2 所示的字符串进行匹配操作（上面为主串，下面为模式串）。

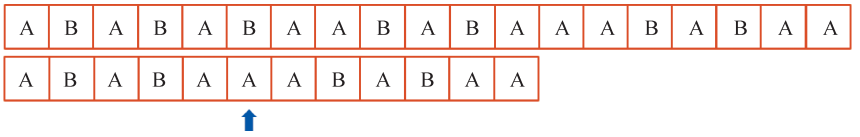

图 5-2　字符串比较

如图 5-3 所示，当进行第一轮比较时，与 BF 算法类似，将主串与模式串左端对齐，指针（箭头）所示位置发生了不匹配。在模式串中，发生不匹配的字符之前，用方框标识两个子串“ABA”，我们分别将其称为“公共前缀”（蓝）和“公共后缀”（绿）。

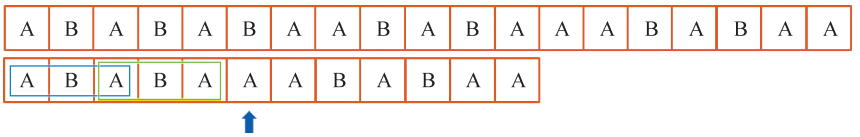

图 5-3　公共前后缀

接下来，移动模式串，将“公共前缀”移动到“公共后缀”所在的位置，移动后如图 5-4 所示。

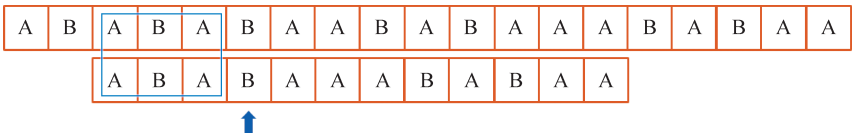

图 5-4　将“公共前缀”移动到“公共后缀”

通过观察可以发现，因为公共前后缀是相同的，所以将“公共前缀”移动到“公共后缀”所在的位置后，指针左侧上下的子串依然是匹配的。这样我们就在确保不跳过可能存在的匹配的情况下，一次性移动了两个字符，打破了暴力匹配算法中每次后移一个字符的限制。安全移动过程如图 5-5 所示。

“老师，这样移动真的不会遗漏可能存在的匹配吗？”

“当然不会，小萌可以自己尝试下看看。”

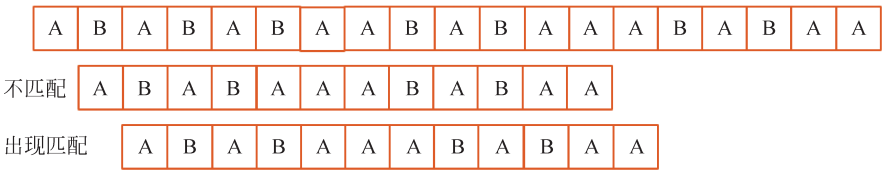

图 5-5　安全移动

“老师，果然如您所说，没有出现遗漏匹配的情况，将‘公共前缀’移动到‘公共后缀’的位置刚刚好！”

所以，我们可以总结出 KMP 算法中模式串的移动规则为：当进行比较时，模式串中某个字符 X 发生不匹配时，只要找到字符 X 左侧子串中的“公共前缀”和“公共后缀”，将“公共前缀”移动到“公共后缀”所在位置，如此就实现了模式串的跨越式移动。

“老师，上面例子中的‘公共前缀’和‘公共后缀’是怎么确定的啊？”

“小帅问得好！现在我就来介绍下‘公共前缀’和‘公共后缀’的确定方法。”

公共前后缀的确定方法如下。

（1）前缀是指除最后一个字符外，一个字符串的全部头部组合。

（2）后缀是指除第一个字符外，一个字符串的全部尾部组合。

（3）“公共前后缀”是指模式串中前缀和后缀所共有的字符元素。

例如，下列各字符串的公共前后缀为：

“A”无前后缀，因为规定前后缀长度需小于子串长度，所以只有一个字符的字符串公共前后缀的长度为 0。

“AB”的前缀为 [A]，后缀为 [B]，共有元素为空，公共前后缀长度为 0。

“ABA”的前缀为 [A，AB]，后缀为 [BA，A]，共有元素为“A”，公共前后缀长度为 1。

“ABAB”的前缀为 [A，AB，ABA]，后缀为 [BAB，AB，B]，共有元素为“AB”，公共前后缀长度为 2。

“ABABA”的前缀为 [A，AB，ABA，ABAB]，后缀为 [BABA，ABA，BA，A]，共有元素为“A”“ABA”。出现了两对共有元素，长度分别为 1、3，此时我们选择最长的公共前后缀“ABA”，因此其公共前后缀长度为 3。

在刚才的例子中我们找的并不仅仅是“公共前后缀”，准确地讲，应该是“最长公共前后缀”。因为公共前后缀可能有多个，而我们找的是其中最长的那一对。

其余的子串也按类似的方法进行处理。

当模式串中某一个字符发生不匹配时，仅根据“不匹配字符自身”和“最长公共前后缀”就可以单方面确定下一次移动的距离，而这个过程与主串并无关系。所以，可以在开始匹配操作之前就计算好每个字符发生不匹配时的移动方案，并形成如表 5-1 所示的移动方案表。

表 5-1　发生不匹配时的移动方案

字　符	移 动 方 案
A	最长公共前后缀长度为 0，1 号位与主串下一位比较
B	最长公共前后缀长度为 0，1 号位与主串当前位比较
A	最长公共前后缀长度为 0，1 号位与主串当前位比较
B	最长公共前后缀长度为 1，2 号位与主串当前位比较
A	最长公共前后缀长度为 2，3 号位与主串当前位比较
A	最长公共前后缀长度为 3，4 号位与主串当前位比较
A	最长公共前后缀长度为 1，2 号位与主串当前位比较
B	最长公共前后缀长度为 1，2 号位与主串当前位比较
A	最长公共前后缀长度为 2，3 号位与主串当前位比较
B	最长公共前后缀长度为 3，4 号位与主串当前位比较
A	最长公共前后缀长度为 4，5 号位与主串当前位比较
A	最长公共前后缀长度为 5，6 号位与主串当前位比较

这个移动方案是怎么得到的呢？我们来解读一下。

为了便于描述，先为模式串中的每个字符进行编号（从 1 开始），如图 5-6 所示。

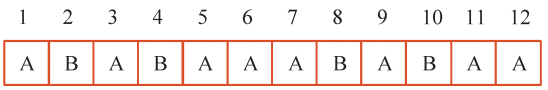

图 5-6　为模式串字符编号（从 1 开始）

假设发生如图 5-7 所示的情况，即模式串中的第 4 个字符 B 发生不匹配。

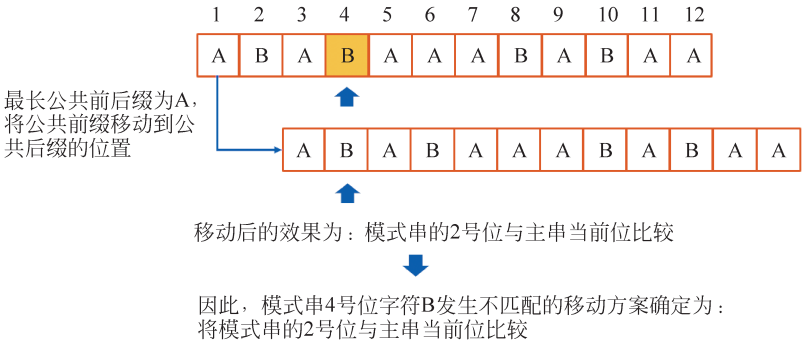

图 5-7　计算移动方案

为了确定其移动方式，根据算法规则，需要先确定其左侧子串“ABA”中的最长公共前后缀，子串“ABA”的最长公共前后缀为“A”，随后将公共前缀移动到公共后缀所在位置，移动后的效果为：模式串的 2 号位（即第二个字符 B）与主串当前位对齐。因此，只要当 4 号位的字符 B 发生不匹配时，我们都按照同样的方案进行移动即可。对模式串中的每个字符重复以上计算过程，最终就得到了移动方案表。

通过观察可以发现如下规律：如果最长公共前后缀的长度为 n，就可以得到“将模式串 n+1 号位与主串当前位比较”的规则。

接下来，我们将模式串的第一个字符的值标记为 0（该字符比较特殊，其值通常直接赋值为 0），其余字符按移动方案中开头数字编号，如模式串中的第二个字符“B”，其移动方案为“1 号位与主串当前位比较”，所以其值标记为 1，以此类推，我们就可以得到值与字符下标的对应关系表，该表通常称为 Next 表，如表 5-2 所示。

表 5-2　Next 表

字符	A	B	A	B	A	A	A	B	A	B	A	A
下标	1	2	3	4	5	6	7	8	9	10	11	12
值	0	1	1	2	3	4	2	2	3	4	5	6

Next 表指明了当特定字符发生不匹配时，模式串要移动到的位置。因此在算法开始正式的匹配运算前，程序需要先生成 Next 表，这个阶段被称为“预处理阶段”。

2. 代码实现

KMP 算法的实现代码如下。

```
'''
    构造临时 next 数组
'''
def getNext(pattern):
    # 初始化 next 数组
    next_list = [0 for i in range(len(pattern))]
    # 初始化下标，j 指向模式串
    #t 记录位置，便于 next 数组赋值
    j = 1
    t = 0
    while j < len(pattern):
        # 第一次比较：t 等于 0，直接进行赋值，初始化 1 号位移动方案，每次
        # 长度自增 1
        # 后续比较：判断字符是否相等，Python 数组下标从 0 开始，因此均减 1
        if t == 0 or pattern[j - 1] == pattern[t - 1]:
            # 长度加 1
            next_list[j] = t + 1
            j += 1
            t += 1
        else:
            # 字符不相等，t 回溯
            t = next_list[t - 1]
    return next_list

'''
    KMP 字符串匹配的主函数
    若匹配成功，返回模式串在主串中开始位置的下标
```

```
    若匹配不成功，返回 -1
'''
def KMP(text, pattern):
    next = getNext(pattern)
    # 主串计数
    i = 0
    # 模式串计数
    j = 0
    while i < len(text) and j < len(pattern):
        if (text[i] == pattern[j]):
            i += 1
            j += 1
        elif j != 0:
            # 调用 next 数组
            j = next[j - 1]
        else:
            # 不匹配主串指针后移
            i += 1
    if j == len(pattern):
        return i - len(pattern)
    else:
        return -1


if __name__ == "__main__":
    text = 'abcxabcdabcdabcy'
    pattern = 'abcdabcy'
    out = KMP(text, pattern)
    print(out)
```

【问题 5-2】 使用 KMP 算法实现如下字符串匹配程序：给定一个主串