串的定义

串（string）是有0~n个字符组成的有限序列，一般记为

$$S =' a_1 a_2 \ldots a_n' (n \geq 0) \tag{1}$$

`S` 是字符串的名称，a_i 可以是字母数字或其他字符。

其中，串中任意连续的字符组成的子序列称为该串的 **子串。**

字符在串中的位置 通常是该字符在序列中的序号

子串在主串中的位置 指的是子串第一个字符在主串中的位置

串的模式匹配

子串的定位操作通常称之为串的模式匹配，它的目的是要获取子串在主串的位置。

一般的，我们想到的是一种暴力破解的方法

也就是说，从主串 `S` 的第 `pos` 个字符起，与模式串的一个字符比较，若相等，则继续逐个比较主串和模式的后续字符；否则，从主串的下一个字符起，重新和模式的字符比较，以此类推，直至成功或失败。

```
int IndexSubstring(string s, string sub, int pos) {
    int i = pos;
    int j = 0;
    while(i < s.size() && j < sub.size()) {
        if(s[i]==sub[j]) {
            i++;
            j++;
        } else {
            i = i - j + 1;
            j = 0;
        }
    }
    if(j >= sub.size()) return i - sub.size() + 1;  // 查找成功
    else return 0; // 查找失败
}
```

但这种暴力破解的算法的最坏时间复杂度为O(mn)，m和n分别是主串和模式串的长度。

KMP算法——改进的模式匹配算法

在回顾上面暴力破解的算法，我们可以得到，暴力算法每一次回溯都是回到模式串最初的起点，事实上，这一步是可以改进的。

【KMP算法】利用比较过的信息，`i` 指针不需要回溯，仅将子串向后滑动一个合适的位置，并从这个位置开始和主串比较，这个合适的位置仅与 **子串本身结构有关，与主串无关。**

【KMP算法】实际上是一种【备忘录设计模式】，下面通过几个步骤或许可以让你了解这个算法的使用。

我们以主串 `s='ababcabcacbab'`，模式串 `sub='abcac'`

1.获取模式串的 next 数组（备忘录）

子串	前缀	后缀	前后缀相同的最长
a	-	-	0
ab	a	b	0
abc	a,ab	c,bc	0
abca	a,ab,abc	a,ca,bca	1
abcac	a,ab,abc,abca	c,ac,cac,bcac	0

所以 `模式串sub` 的 `next数组` 为

s	a	b	c	a	c
next	0	0	0	1	0

2.匹配过程

要满足一个公式：**移动位数 = 已经匹配的位数 - 对应的部分匹配值**

第一趟：发现 `a` 与 `c` 不配

a	b	a	b	c	a	b	c	a	c	b	a	b
a	b	c										

但是前面两个字符 `'ab'` 是匹配的，查表可知，最后一个匹配字符 `b` 对应匹配部分值为 `0`，按公式，计算 $2-0=2$，所以，模式串向后移动 `2` 位。

第2趟：模式串向后移动 `2` 位后，发现 `b` 与 `c` 不配

a	b	a	b	c	a	b	c	a	c	b	a	b
		a	b	c	a	c						

但是前面两个字符 `'abca'` 是匹配的，查表可知，最后一个匹配字符 `a` 对应匹配部分值为 `1`，按公式，计算 $4-1=3$，所以，模式串向后移动 `3` 位。

第3趟：模式串向后移动 `3` 位后，匹配成功

a	b	a	b	c	a	b	c	a	c	b	a	b
					a	b	c	a	c			

使用【KMP算法】后，时间复杂度变为了O(m+n)

KMP原理

已知公式：移动位数 = 已经匹配的位数 - 对应的部分匹配值

改为代码：

$$move = (j-1) - next[j-1] \tag{2}$$

优化公式，如果我们将 `next数组` 向后挪一位，就不需要 `next[j-1]` ，只需要直接使用 `next[j]` 即可

s	a	b	c	a	c
next	0	0	0	1	0
右移	-1	0	0	0	1

于是上述公式（2）既可以改为

$$move = (j-1) - next[j] \tag{3}$$

此时，得到串移动的公式

$$j = j - move = j - ((j-1) - next[j]) = next[j] + 1 \tag{4}$$

进一步优化，我们可以将右移后是 `next数组` 总体+1，那么，最后的数组就变成了

$$j = next[j] \tag{5}$$

s	a	b	c	a	c
next	0	0	0	1	0
右移	-1	0	0	0	1
右移+1	0	1	1	1	2

最后一行的意思是，把模式串的第 `next[j]` 个值，移动 `j` 这个位置。或者说，从模式串的第 `next[j]` 个数组开始查找，主串继续移动即可。

2.使用了改进后的next数组再匹配过程

第一趟：发现 `a` 与 `c` 不配

a	b	a	b	c	a	b	c	a	c	b	a	b
a	b	c										

在 `c（pos=2）` 这个位置匹配失败，查表得，`c` 的对应值是 `1` ，也就是将模式串的第一个字符 `（a）` 移到 `c（pos=2）` 这个位置

第2趟：模式串向后移动 `2` 位后，发现 `b` 与 `c` 不配

a	b	a	b	c	a	b	c	a	c	b	a	b
		a	b	c	a	c						

在 `c（pos=6）` 这个位置匹配失败，查表得，`c` 的对应值是 `2` ，也就是将模式串的第二个字符 `（b）` 移到 `c（pos=6）` 这个位置

第3趟：模式串向后移动 `3` 位后，匹配成功

a	b	a	b	c	a	b	c	a	c	b	a	b
					a	b	c	a	c			

参考代码

获取next数组

```
void getNext(string sub, int next) {
    int i = 0;
    int j = 0;
    next[0] = 0;
    while(i<sub.size()) {
        if(j == 0|| s[i]==s[j]){
            i++;
            j++;
            next[i] = j;
        } else {
            j = next[j];
        }
    }
}
```

【KMP算法】

```
int Index(string s, string sub, int next[], inr pos) {
    int i = pos;
    int j = 0;
    while(i<s.size()&&j<sub.size()) {
```

```
        if(j==0||s[i]==sub[j]){
            i++;
            j++;
        }
        else {
            j = next[j];    // 模式串右移
        }

        if(j >= sub.size()) {
            return i - sub.size() + 1;
        }
        eles {
            return 0;
        }

    }
}
```