KMP算法

【题目】

给定两个字符串str和match，长度分别为N和M。实现一个算法，如果字符串str中含有字串match，则返回match在str中的开始位置，不含有则返回-1。

【举例】

str=“acbc”，match=“bc”。返回2。

str=“acbc”，match=“bcc”。返回-1。

【要求】

如果match的长度大于str长度(M>N)，str必然不会含有match，可直接返回-1。但如果N>=M，要求算法复杂度O(N)。

【难度】

将 ★★★★

【解答】

本文是想重点介绍一下KMP算法，该算法是由Donald Knuth、Vaughan Pratt和James H. Morris于1977年联合发明的。在介绍KMP算法之前，我们先来看看普通解法怎么做。

最普通的解法是从左到右遍历str的每一个字符，然后看看如果以当前字符作为第一个字符出发是否匹配出match。比如str=“aaaaaaaaaaaaaaaaab”，match=“aaaab”。从str[0]出发，开始匹配。匹配到str[4]==‘a’时发现和match[4]==‘b'不一样，所以匹配失败。说明从str[0]出发是不行的。从str[1]出发，开始匹配。匹配到str[5]=='a'时发现和match[4]=='b'不一样，所以匹配失败。说明从str[1]出发是不行的。从str[2..12]出发，都会一直失败。从str[13]出发，开始匹配。匹配到str[17]=='b'时发现和match[4]=='b'一样，match已经全部匹配完，说明匹配成功，返回13。普通解法的时间复杂度较高，从每个字符出发时，匹配的代价都可能是O(M)，那么一共有N个字符，所以整体的时间复杂度为O(N*M)。普通解法时间复杂度这么高，是因为每次遍历到一个字符，检查工作相当于从无开始，之前的遍历检查不能优化当前的遍历检查。

下面介绍KMP算法是如何快速的解决字符串匹配问题的。

1，首先生成match字符串的nextArr数组，这个数组的长度与match字符串的长度一样，nextArr[i]的含义是在match[i]之前的字符串match[0..i-1]中，必须以match[i-1]结尾的后缀子串(不能包含match[0])与必须以match[0]开头的前缀子串(不能包含match[i-1])最大匹配长度是多少。这个长度就是nextArr[i]的值。比如，match=“aaaab"这个字符串，nextArr[4]的值该是多少呢？match[4]=='b'，所以它之前的字符串为"aaaa"，根据定义这个字符串的后缀子串和前缀子串最大匹配为"aaa"。也就是当后缀子串等于match[1..3]=="aaa"，前缀子串等于match[0..2]=="aaa"的时候，这时前缀和后缀不仅相等，而且所有前缀和后缀的可能性中最大的匹配。所以nextArr[4]的值等于3。再比如，match="abc1abc1"这个字符串，nextArr[7]的值该是多少呢？match[7]=='1'，所以它之前的字符串为"abc1abc"，根据定义这个字符串的后缀子串和前缀子串最大匹配为"abc"。也就是当后缀子串等于match[4..6]=="abc"，前缀子串等于match[0..2]=="abc"的时候，这时前缀和后缀不仅相等，而且所有前缀和后缀的可能性中最大的匹配。所以nextArr[7]的值等于3。关于如何快速的得到nextArr数组的问题，我们在把KMP算法大概过程介绍完毕之后再详细的说明，接下来先看看如果有了match的nextArr数组，如何加速str和match的匹配过程。

2，假设从str[i]字符出发时，匹配到j位置的字符发现与match中的字符不一致，也就是说str[i]与match[0]一样，并且从这个位置开始一直可以匹配，即str[i..j-1]与match[0..j-i-1]一样，直到发现str[j]!=match[j-i]，匹配停止。如图1-1。

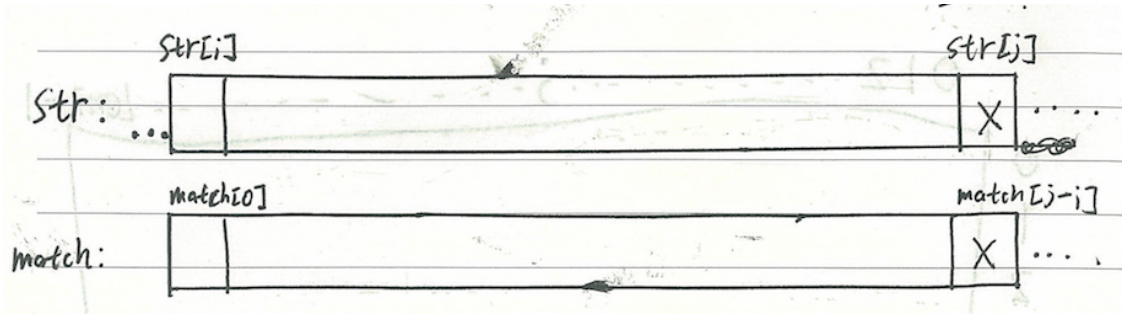

图1-1

因为现在已经有了match字符串的nextArr数组，nextArr[j-i]的值表示了match[0..j-i-1]这一段字符串前缀与后缀的最长匹配。假设前缀是图1-2中的a区域这一段，后缀是图1-2中的b区域这一段，再假设a区域的下一个字符为match[k]。如图1-2。

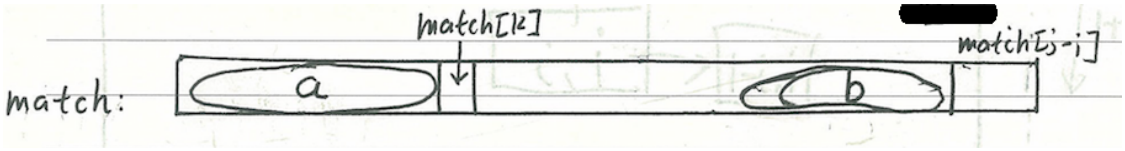

图1-2

那么下一次的匹配检查，不再像普通解法那样退回到str[i+1]重新开始与match[0]的匹配过程，而是直接让str[j]与match[k]进行匹配检查。如图1-3。

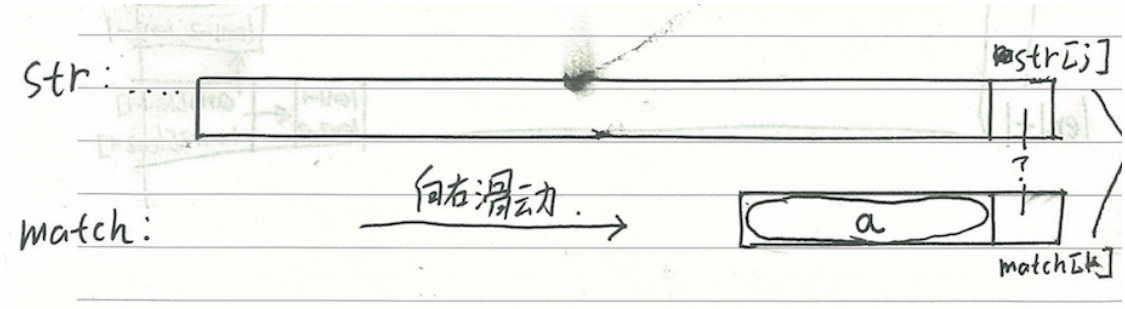

图1-3

如图1-3，在str中要匹配的位置仍是j，而不进行退回。对match来说，相当于向右滑动，让match[k]滑动到与str[j]同一个位置上，然后进行后续的匹配检查。普通解法str要退回到i+1位置，然后让str[i+1]与match[0]进行匹配，而我们的解法在匹配的过程中一直进行这样的滑动匹配的过程，直到在str的某一个位置把match完全匹配完，就说明str中有match。如果match滑到最后也没匹配出来，就说明str中没有match。那么为什么这样做是正确的呢？如图1-4。

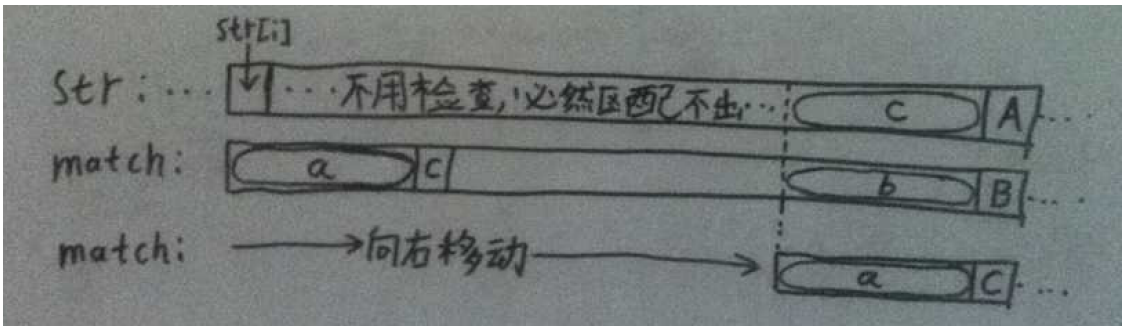

图1-4

在图1-4中，匹配到A字符和B字符才发生的不匹配，所以c区域等于b区域，b区域又与a区域相等(因为nextArr的含义如此)，所以c区域和a区域是不需要检查的，必然会相等。所以直接把字符C滑到字符A的位置开始检查即可。其实这个过程相当于是从str的c区域的第一个字符重新开始的匹配过程(c区域的第一个字符和match[0]匹配，并往右的过程)，只不过因为c区域与a区域一定相等，所以省去了这个区域的匹配检查而已，直接从字符A和字符C往后继续匹配检查。读者看到这里肯定会问，为什么开始的字符从str[i]直接跳到c区域的第一个字符呢？中间的这一段为什么是”不用检查”的区域呢？因为在这个区域上，从任何一个字符出发都肯定匹配不出match，下面还是图解来解释这一点。如图1-5。

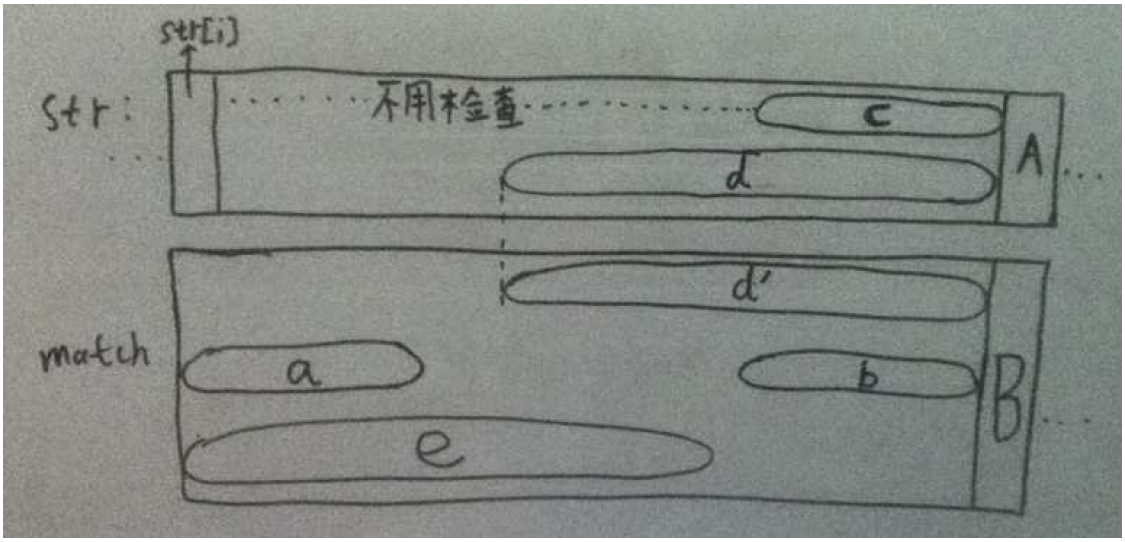

图1-5

图1-5中，假设d区域开始的字符是“不用检查”区域的其中一个位置，如果从这个位置开始能够匹配出match，那么毫无疑问，起码整个d区域应该和从match[0]开始的e区域匹配，即d区域与e区域长度一样，且两个区域的字符都相等。同时我们注意到，d区域比c区域大，e区域比a区域大。如果这种情况发生了，假设d区域对应到match字符串中是d’区域，也就是字符B之前的字符串的后缀，而e区域本身就是match的前缀，所以对于match来说，相当于找到了B这个字符之前的字符串(match[0..j-i-1])的一个更大的前缀与后缀匹配，一个比a区域和b区域更大的前缀后缀匹配，e区域和d’区域。这与nextArr[j-i]的值是自相矛盾的，因为nextArr[j-i]的值代表的含义就是match[0..j-i-1]字符串上最大的前缀与后缀匹配长度。所以如果match字符串的nextArr数组计算正确，这种情况绝不会发生，也就是说根本不会有更大的d’区域和e区域，所以d区域与e区域也必然不会相等。

匹配过程分析完毕，我们知道，str中的匹配的位置是不退回的，match则一直向右滑动，如果在str中的某个位置完全匹配出match，整个过程停止。否则match滑到str的最右侧过程也停止，所以滑动的长度最大为N，所以时间复杂度为O(N)。匹配的全部过程参看如下代码中的getIndexOf方法。

```
public int getIndexOf(String s, String m) {
    if (s == null || m == null || m.length() < 1 || s.length() < m.length()) {
        return -1;
    }
    char[] ss = s.toCharArray();
    char[] ms = m.toCharArray();
    int si = 0;
    int mi = 0;
    int[] next = getNextArray(ms);
    while (si < ss.length && mi < ms.length) {
        if (ss[si] == ms[mi]) {
                si++;
                mi++;
        } else if (next[mi] == -1) {
```

```
                si++;
            } else {
                mi = next[mi];
            }
        }
        return mi == ms.length ? si - mi : -1;
    }
```

最后我们需要解释如何快速得到match字符串的nextArr数组，并且要证明得到nextArr数组的时间复杂度为O(M)。对于match[0]来说，它的之前没有字符，所以nextArr[0]规定为-1。对于match[1]来说，它的之前有match[0]，但nextArr数组的定义要求任何子串的后缀不能包括第一个字符(match[0])，所以match[1]之前的字符串，只有长度为0的后缀字符串，所以nextArr[1]为0。之后对于match[i](i>1)来说，求解过程如下：
1，因为是左到右依次求解nextArr，所以在求解nextArr[i]时，nextArr[0..i-1]的值都已经求出了。假设match[i]字符为图1-6中的A字符，match[i-1]为图1-6中的B字符，如图1-6。

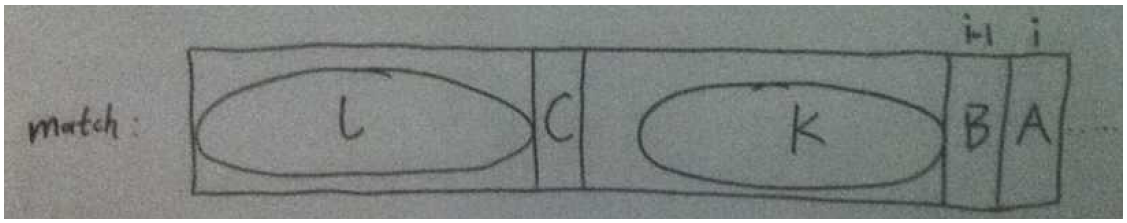

图1-6

通过nextArr[i-1]的值可以知道B字符前的字符串的最长前缀与后缀匹配区域，图1-6中的l区域为最长匹配的前缀子串，k区域为最长匹配的后缀子串，图1-6中字符C为l区域之后的字符。然后看看字符C与字符B是否相等。

2，如果字符C与字符B相等，那么A字符之前的字符串的最长前缀与后缀匹配区域就可以确定了，前缀子串为l区域+C字符，后缀子串为k区域+B字符，即nextArr[i]=nextArr[i-1]+1。

3，如果字符C与字符B不相等，就考察字符C之前的前缀后缀匹配情况，假设字符C是第cn个字符(match[cn])，那么nextArr[cn]就是其最长前缀后缀匹配长度，如图1-7。

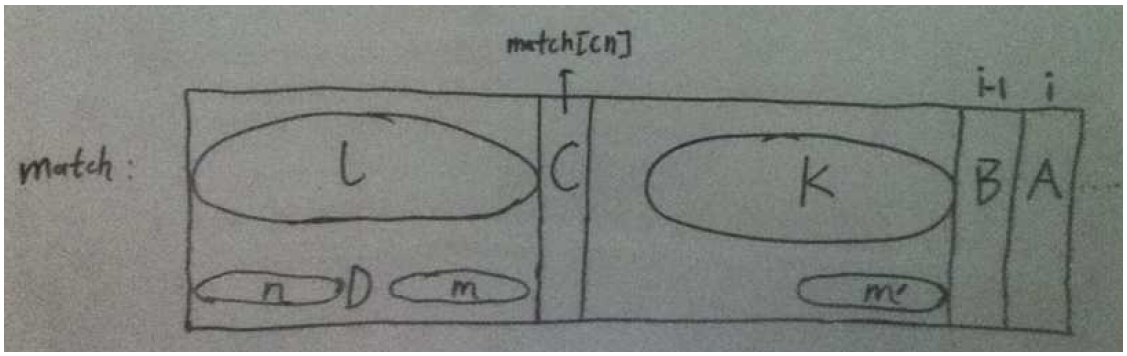

图1-7

图1-7中，m区域和n区域分别是字符C之前的字符串的最长匹配的后缀与前缀区域，这是通过通过nextArr[cn]的值确定的，当然两个区域是相等的，m'区域为k区域最右的区域且长度与m区域一样，因为k区域和l区域是相等的，所以m区域和m'区域也相等，字符D为n区域之后的一个字符，接下来比较字符D是否与字符B相等。

1）如果相等，A字符之前的字符串的最长前缀与后缀匹配区域就可以确定了，前缀子串为n区域+D字符，后缀子串为m'区域+B字符，则令nextArr[i]=nextArr[cn]+1;

2）如果不等，继续往前跳到字符D，之后的过程与跳到字符C类似，一直进行这样的跳过程，跳的每一步都会有一个新的字符拿出来和B比较(就像C字符和D字符一样)，只要

有相等的情况，那nextArr[i]的值就能确定。

4，如果向前跳到最左位置即match[0]的位置，此时nextArr[0]==-1，说明字符A之前的字符串不存在前缀后缀匹配的情况，则令nextArr[i]=0。用这种不断向前跳的方式可以算出正确的nextArr[i]的值的原因，还是因为每跳到一个位置cn，nextArr[cn]的意义就是表示它之前字符串的最大匹配长度。求解nextArr数组的具体过程请参看如下代码中的getNextArray方法，先看代码然后分析这个过程为什么时间复杂度为O(M)。

```
public int[] getNextArray(char[] ms) {
        if (ms.length == 1) {
                return new int[] { -1 };
        }
        int[] next = new int[ms.length];
        next[0] = -1;
        next[1] = 0;
        int pos = 2;
        int cn = 0;
        while (pos < next.length) {
                if (ms[pos - 1] == ms[cn]) {
                        next[pos++] = ++cn;
                } else if (cn > 0) {
                        cn = next[cn];
                } else {
                        next[pos++] = 0;
                }
        }
        return next;
}
```

getNextArray方法中的while循环就是求解nextArr数组的过程，现在证明这个循环发生的次数不会超过2M这个数量。先来看看两个量，一个为pos量，一个为(pos-cn)的量。对于pos量来说，从2开始又必然不会大于match的长度，即pos<M。对于(pos-cn)量来说，pos最大为M-1，cn最小为0，所以(pos-cn)<=M。
循环的第一个逻辑分支会让pos的值增加，(pos-cn)的值不变。循环的第二个逻辑分支为cn向左跳的过程，所以会让cn减小，pos值在这个分支中不变，所以(pos-cn)的值会增加。循环的第三个逻辑分支会让pos的值增加，(pos-cn)的值也增加。如下表所示：

	pos	pos-cn
循环的第一个逻辑分支	增加	不变
循环的第二个逻辑分支	不变	增加
循环的第三个逻辑分支	增加	增加

因为pos+(pos-cn)<2M，又有上表的关系，所以循环发生的总体次数小于pos量和(pos-cn)量的增加次数，也必然小于2M，证明完毕。

所以整个KMP算法的复杂度为O(M)(求解nextArr数组的过程)+O(N)(匹配的过程)，因为有N>=M，所以时间复杂度为O(N)。