sofu6

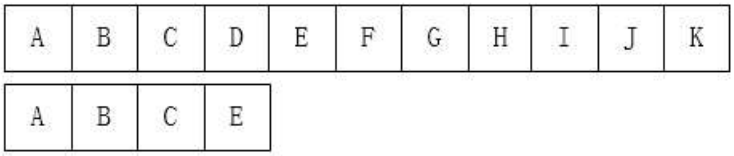
KMP算法详解-彻底清楚了(转载+部分原创)

引言

KMP算法指的是字符串模式匹配算法，问题是：在主串T中找到第一次出现完整子串P时的起始位置。该算法是三位大牛：D.E.Knuth、J.H.Morris和V.R.Pratt同时发现的，以其名字首字母命名。在网上看了不少对KMP算法的解析，大多写的不甚明了。直到我看到一篇博客的介绍，看完基本了解脉络，本文主要是在其基础上，在自己较难理解的地方进行补充修改而成。该博客地址为：https://www.cnblogs.com/yjiyjige/p/3263858.html，对作者的明晰的解析表示感谢。

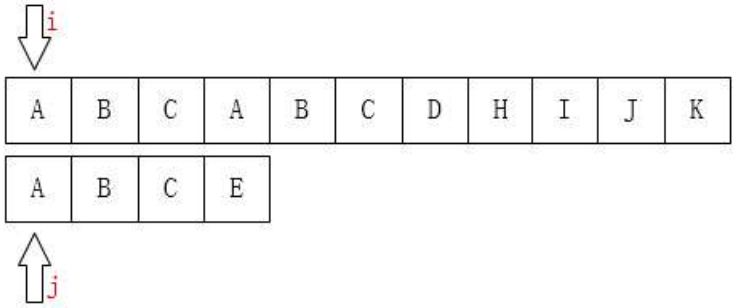
1. 一般的解法

KMP算法要解决的问题就是在字符串（也叫主串）中的模式（pattern）定位问题。说简单点就是我们平时常说的关键字搜索。模式串就是关键字（接下来称它为P），如果它在一个主串（接下来称为T）中出现，就返回它的具体位置，否则返回-1（常用手段）。

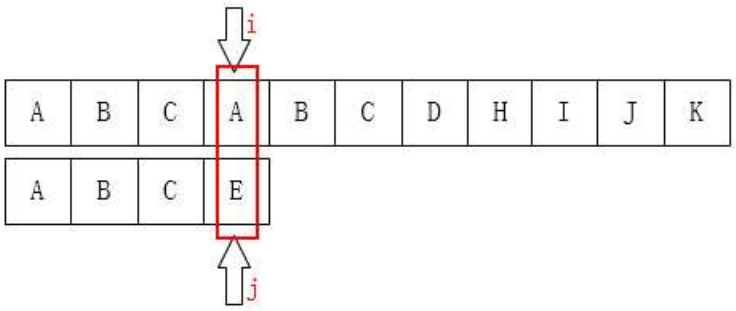


首先，对于这个问题有一个很直接的想法：从左到右一个个匹配，如果这个过程中有某个字符不匹配，就跳回去，将模式串向右移动一位。这有什么难的？

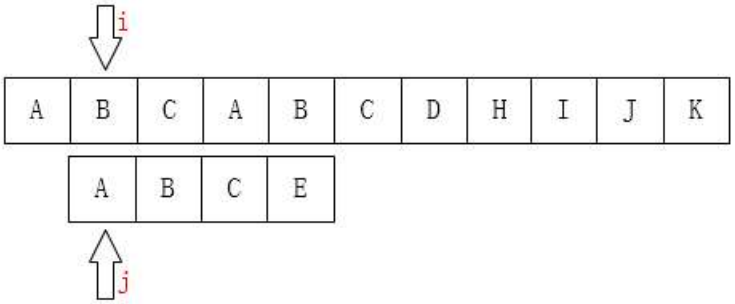
我们可以这样初始化：



之后我们只需要比较i指针指向的字符和j指针指向的字符是否一致。如果一致就都向后移动，如果不一致，如下图：



A和E不相等，那就把i指针移回第1位（假设下标从0开始），j移动到模式串的第0位，然后又重新开始这个步骤：



基于这个想法我们可以得到以下的程序:

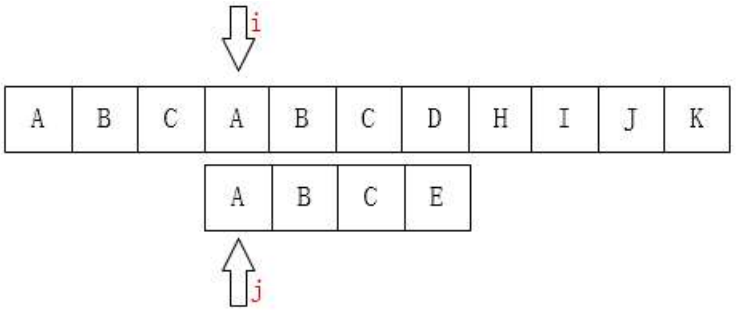
```
/**
 * 暴力破解法
 * @param ts 主串
 * @param ps 模式串
 * @return 如果找到，返回在主串中第一个字符出现的下标，否则为-1
 */
public static int bf(String ts, String ps) {
    char[] t = ts.toCharArray();
    char[] p = ps.toCharArray();
    int i = 0; // 主串的位置
    int j = 0; // 模式串的位置
    while (i < t.length && j < p.length) {
        if (t[i] == p[j]) { // 当两个字符相同，就比较下一个
            i++;
            j++;
        } else {
            i = i - j + 1; // 一旦不匹配，i后退
            j = 0; // j归0
        }
    }
    if (j == p.length) {
        return i - j;
    } else {
        return -1;
    }
}
```

上面的程序是没有问题的，但不够好！（想起我高中时候数字老师的一句话：我不能说你错，只能说你不对~~~）

注意：该算法程序很简单，非常好理解，请认真看完，因为后面的算法是在该算法基础上修订的。

2.如果人眼来优化的话，怎样处理

参考上面的算法，**我们串中的位置指针i,j来说明，第一个位置下标以0开始**，我们称为第0位。下面看看，如果是人为来寻找的话，肯定不会再把i移动回第1位，**因为主串匹配失败的位置(i=3)前面除了第一个A之外再也没有A了**，我们为什么能知道主串前面只有一个A？**因为我们已经知道前面三个字符都是匹配的！（这很重要）**。移动过去肯定也是不匹配的！有一个想法，**i可以不动，我们只需要移动j**即可，如下图：

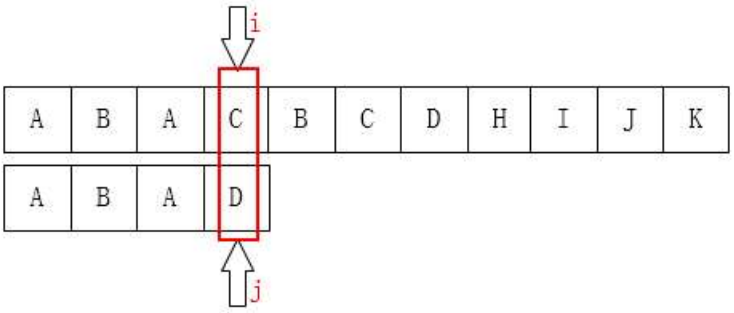


上面的这种情况还是比较理想的情况，我们最多也就多比较了再次。但假如是在主串"SSSSSSSSSSSSSA"中查找"SSSSB"，比较到最后一个才知道不匹配，然后i回溯，这个的效率是显然是最低的。

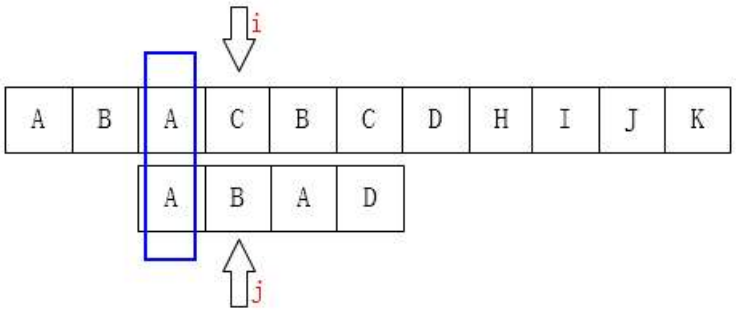
大牛们是无法忍受"暴力破解"这种低效的手段的，于是他们三个研究出了KMP算法。其思想就如同我们上边所看到的一样："**利用已经部分匹配这个有效信息，保持i指针不回溯，通过修改j指针，让模式串尽量地移动到有效的位置。**"

所以，整个KMP的重点就在于**当某一个字符与主串不匹配时，我们应该知道j指针要移动到哪**？

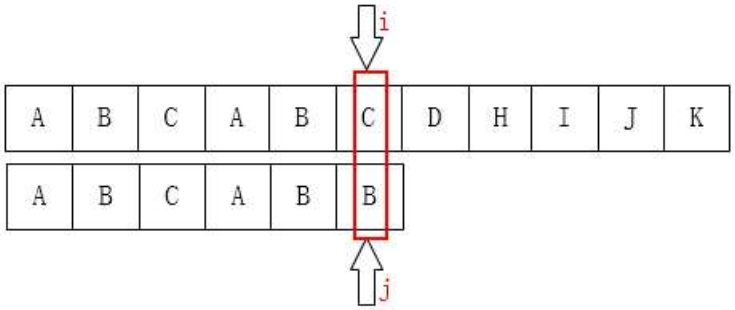
接下来我们自己来发现j的移动规律：



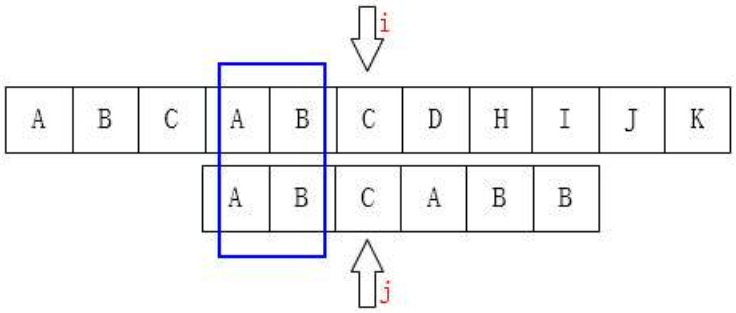
如图：C和D不匹配了，我们要把j移动到哪？显然是第1位。为什么？因为前面有一个A相同啊：



如下图也是一样的情况：



可以把j指针移动到第2位，因为前面有两个字母是一样的：

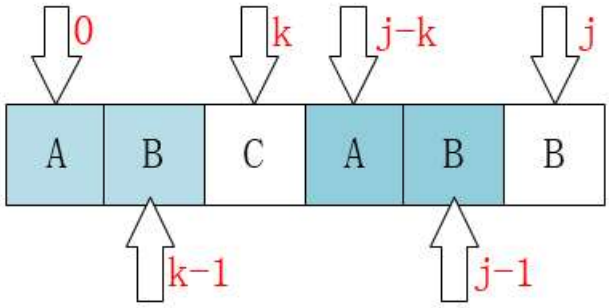


至此我们可以大概看出一点端倪，当匹配失败时，j要移动的下一个位置k。存在着这样的性质：**最前面的k个字符和j之前的最后k个字符是一样的。**

如果用数学公式来表示是这样的

$$P[0 \sim k-1] == P[j-k \sim j-1]$$

这个相当重要，如果觉得不好记的话，可以通过下图来理解：



弄明白了这个就应该可能明白为什么可以直接将j移动到k位置了。

因为：

当T[i] != P[j]时

有T[i-j ~ i-1] == P[0 ~ j-1]

由P[0 ~ k-1] == P[j-k ~ j-1]

必然：T[i-k ~ i-1] == P[0 ~ k-1]

原文说公式很无聊，但我觉得这样简单的公式就能清楚表达我们想说的含义，实在是幸甚。这个公式小学生都能看懂的，真的，我教三年级的娃就告诉她这个了。无非就是连续的序列的首尾下标和连续序列长度三者之间的关系。设首下标为head，尾下标为tail，序列长度为len，则公式为：len=tail-head+1；head=tail-len+1；我们head为0，则更简化了：len=tail+1；知道这个了，请一定耐着性子看懂，对我们的理解很有帮助。下面所有的公式都是这个相关的，请都要看懂。

这一段公式证明了我们为什么可以直接将j移动到k而无须再比较前面的k个字符。

补充说明：

该规律是KMP算法的关键，KMP算法是利用待匹配的子串自身的这种性质，来提高匹配速度。该性质在许多其他中版本的解释中还可以描述成：若子串的前缀集和后缀集中，重复的最长子串的长度为k，则下次匹配子串的j可以移动到第k位(下标为0为第0位)。我们将这个解释定义成最大重复子串解释。

这里面的前缀集表示除去最后一个字符后的前面的所有子串集合，同理后缀集指的的是除去第一个字符后的后面的子串组成的集合。举例说明如下：

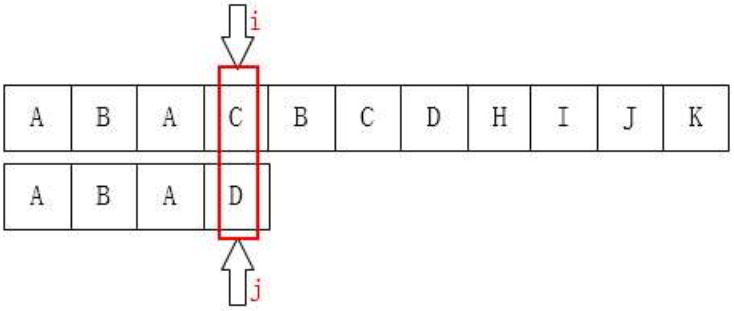
在“aba”中，前缀集就是除掉最后一个字符'a'后的子串集合{a,ab}，同理后缀集为除掉最前一个字符a后的子串集合{a,ba}，那么两者最长的重复子串就是a，k=1；

在“ababa”中，前缀集是{a,ab,aba,abab}，后缀集是{a,ba,aba,baba}，二者最长重复子串是aba，k=3；

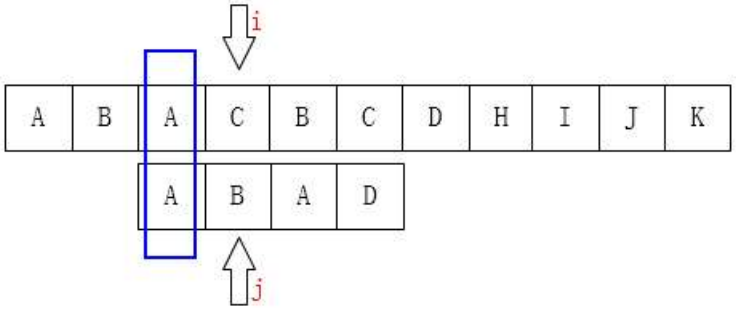
在“abcabcdabc”中，前缀集是{a,ab,abc,abca,abcab,abcabc,abcabcd,abcabcda,abcabcdab}，后缀集是{c,bc,abc,dabc,cdabc,bcdabc,abcdabc,cabcdabc,bcabcdabc}，二者最长重复的子串是“abc”,k=3；

下面我们用这个解释，来再一次手动求解上面的过程：

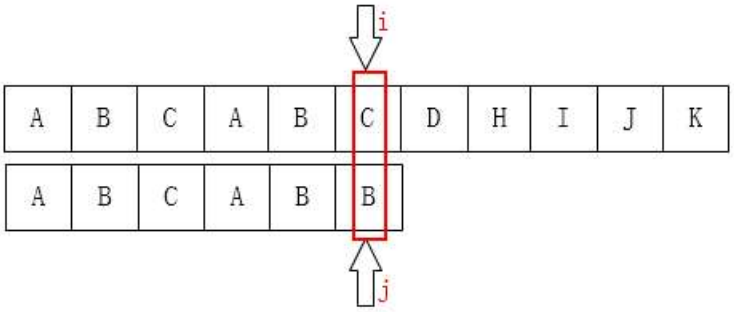
首先如下图所示：



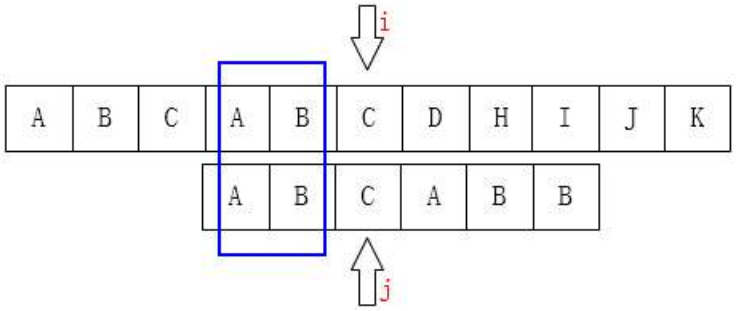
如图：C和D不匹配了，我们要把j移动到哪？j位前面的子串是ABA，该子串的前缀集是{A,AB}，后缀集是{A,BA}，最大的重复子串是A，只有1个字符，所以j移到k即第1位。



再分析下图的情况：



在j位的时候，j前面的子串是ABCAB，前缀集是{A,AB,ABC,ABCA}，后缀集是{B,AB,CAB,BCAB}，最大重复子串是AB，个数是2个字符，因此j移到k即第2位。



上面说的,如果分解成计算机的步骤，则是如下的过程：

1）找出前缀pre，设为pre[0~m]；

2）找出后缀post，设为post[0~n]；

3）从前缀pre里，先以最大长度的s[0~m]为子串，即设k初始值为m,跟post[n-m+1～n]进行比较：

如果相同，则pre[0~m]则为最大重复子串，长度为m，则k=m；

如果不相同，则k=k-1;缩小前缀的子串一个字符，在跟后缀的子串按照尾巴对齐，进行比较，是否相同。

如此下去，直到找到重复子串，或者k没找到。

改天，这里我写个代码说明，怎么找重复子串。

根据上面的求解过程，我们知道子串的j位前面，有j个字符，前后缀必然少掉首尾一个字符，因此重复子串的最大值为j-1，因此知道下一次的j指针最多移到第j-1位。

我为什么要补充上面这段说明，是因为该说明能便于我们理解下面的求解next数组的过程，上面实际也是指出了人工求解next[j]的过程。不知道next[j]为何物没关系，看到下面的定义以后，请到时再绕回来回味就行了。

3.求next数组

好，接下来就是重点了，怎么求这个（这些）k呢？因为在P的每一个位置都可能发生不匹配，也就是说我们要计算每一个位置j对应的k，所以用一个数组next来保存，**next[j] = k，表示当T[i] != P[j]时，j指针的下一个位置。另一个非常有用且恒等的定义，因为下标从0开始的，k值实际是j位前的子串的最大重复子串的长度。请时刻牢记next数组的定义，下面的解释是死死地围绕着这个定义来解释的。**

很多教材或博文在这个地方都是讲得比较含糊或是根本就一笔带过，甚至就是贴一段代码上来，为什么是这样求？怎么可以这样求？根本就没有说清楚。而这里恰恰是整个算法最关键的地方。

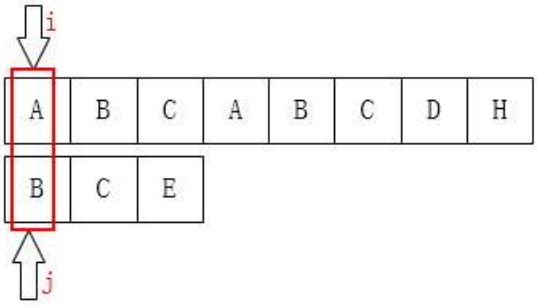
```
1 public static int[] getNext(String ps) {
2
3     char[] p = ps.toCharArray();
4
5     int[] next = new int[p.length];
6
7     next[0] = -1;
8
9     int j = 0;
10
11    int k = -1;
12
13    while (j < p.length - 1) {
14
15        if (k == -1 || p[j] == p[k]) {
16
17            next[++j] = ++k;
18
19        } else {
20
21            k = next[k];
22
```

```
23         }
24
25     }
26
27     return next;
28
29 }
```

这个版本的求next数组的算法应该是流传最广泛的，代码是很简洁。可是真的很让人摸不到头脑，它这样计算的依据到底是什么？

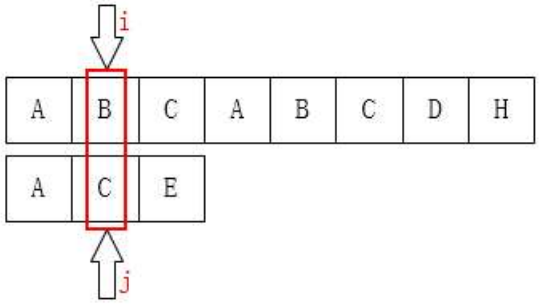
好，先把这个放一边，我们自己来推导思路，现在要始终记住一点，**next[j]的值（也就是k）表示，当P[j] != T[i]时，j指针的下一步移动位置。**

先来看第一个：当j为0时，如果这时候不匹配，怎么办？



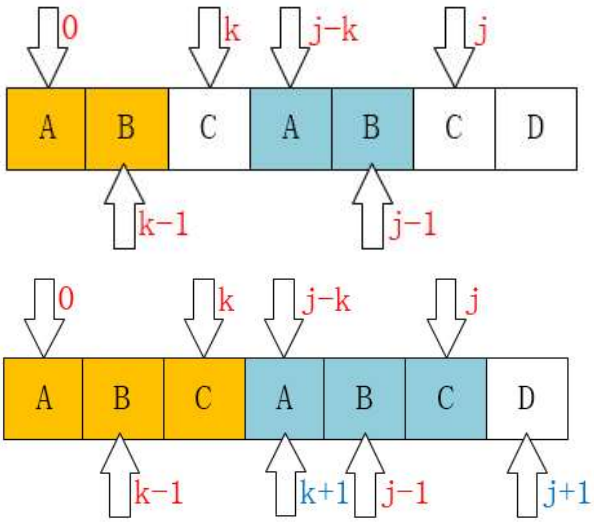
像上图这种情况，**j已经在最左边了，不可能再移动了，这时候要应该是i指针后移。**所以在代码中才会有next[0] = -1;这个初始化。

如果是当j为1的时候呢？



显然，**j指针一定是后移到0位置的。**因为它前面也就只有这一个位置了~~~

下面这个是最重要的，请看如下图：



请仔细对比这两个图。

我们发现一个规律：

当P[k] == P[j]时，

有next[j+1] == next[j] + 1

其实这个是可以证明的：

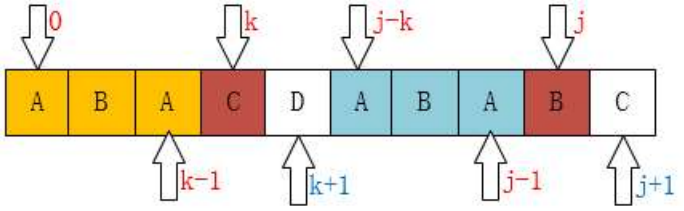
因为在P[j]之前已经有P[0 ~ k-1] == p[j-k ~ j-1]。（next[j] == k）

这时候现有P[k] == P[j]，我们是不是可以得到P[0 ~ k-1] + P[k] == p[j-k ~ j-1] + P[j]。

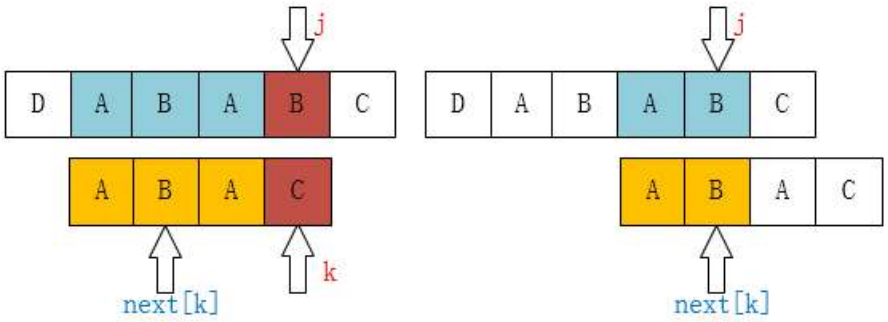
即：P[0 ~ k] == P[j-k ~ j]，即next[j+1] == k + 1 == next[j] + 1。

原文说公式不好懂，看图容易。我觉得，公式实际挺简单的，结合图再把公式耐着性子看懂。实际上，该公式无非是用字母下标代表序列的起始段，描述了前缀和后缀重复相等的一段长度的序列罢了。

那如果P[k] != P[j]呢？比如下图所示：



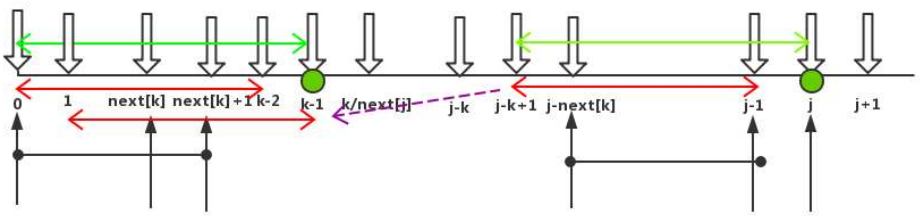
像这种情况，如果你从代码上看应该是这一句：k = next[k];为什么是这样子？你看下面应该就明白了。



现在你应该知道为什么要k = next[k]了吧！像上边的例子，我们已经不可能找到[A，B，A，B]这个最长的后缀串了，但我们还是可能找到[A，B]、[B]这样的前缀串的。所以这个过程像不像在定位[A，B，A，C]这个串，当C和主串不一样了（也就是k位置不一样了），那当然是把指针移动到next[k]啦。

补充说明：看了上面这段的描述，你是否真的理解了P[k]!=P[j]时，是要使用k=next[k]的语句呢？我反正是没弄懂，我总觉得这段else的代码有点反人类，无法理解。实际上，我们的目的是用数学归纳法，来求解next数组的每个值。当前已经求到next[j],接着就应该求解next[j+1]，此时就分两种情况，一种是：重复的字符串个数会增加，即所谓的p[k]=p[j],此时p[j+1]=k+1;即p[++j]=++k;另一种就是不能增加，也就是说P[k]!=P[j],即最大重复子串的长度不能增加了；按照**next[j]的定义，就是当子串的第j位和主串的第i位不一致时，下一次，和主串i位进行比较的子串的j指针的位置。**这个定义还是不太直观，主要是指脑子里不知道是怎样实际操作的，那你回头看看，我上面写的另一个最大重复子串长度的定义，next[j]的值k就是j位之前的子串中，前缀集和后缀集中的最大重复子串的长度。以这个定义我们来尝试在next[j]=k,p[k]!=p[j]时，手动求解next[j+1]的值。

请看下面的图：



当p[j]!=p[k]时我们要找的就是j+1位前面的子串，即p[0~j]的最大重复子串长度。就是说找到一个最长的子串，假设最长重复子串长度为k1，即p[0~k1-1],使得p[0~k1-1]===p[j+1-k1~j],此时k1即为所求的

位置即next[j+1]=k1;因为p[k]!=p[j]了，因此k1最大等于k，即最大可能的重复子串只可能是p[0~k-1]里的子串。此时我们人工求解的话，显然就是从p[0~k-1]里求解最大重复子串。

我们按照第2节介绍的查找最长重复子串的方法：从p[0~k-1]里，第一步，以0位为起始字符先挑选最大子串p[0~k-1]，然后拿着这个子串，尾巴对齐，即看p[k-1]和p[j]对齐，与子串p[j-k+1~j]进行比较，见图中绿色线段；如果线段上每个值都相等了，则找到最大重复子串p[0~k-1];如果不等，则继续缩小线段长度找下去。

下面重点来了，请注意：我们看，在查找最大匹配的过程中，将上面选择的待比较的子串分成两部分：最后一个端点为一部分，前面的一段为一部分；比如上面的第一个选取的最大比较子串的例子：前缀的p[0~k-1]分成两段为p[0~k-2]和p[k-1],和后缀的p[j-k+1～j-1]和p[j]分别比较，即p[0~k-2]和p[j-k+1~j-1]比较，p[k-1]和p[j]比较，见图中的红色线段和绿色圆点；通过这个例子我们知道，只要前面一段能重复且尽可能的长，那么加上最后一个端点这个重复子串也必将是最长的。我们继续分析，因为next[j]已经求出，即p[0~k-1]===p[j-k~j-1],我们可以把上面的第一段的比较进一步转换成，比较p[0~k-2]和p[1~k-1]子串了，见图中紫线箭头指示的漂移；看到没有，这个就是求k位前的子串p[0~k-1]的最大重复子串，很显然不就是求next[k]嘛？！很明显p[0~next[k]-1]就是我们要找的第一个候选最大的重复子串，这也说明了子串p[0~k-2]就不可能是重复子串，也没有尝试比较的必要。因为根据next[j]的定义我们知道，next[k]就是要求的子串为p[0~k-1]的最大重复子串的长度，最大，最大，最大，重要的事说三遍。我们是充分利用了前面k<j时，next[k]已经求出来的条件，减少了子串比较的次数(其实也不叫减少了，那些比较本来就是无效的)；这解释了为什么把k=next[k]。此时，p[0~next[k]-1]和p[j-next[k]~j-1]子串已经恒等了，我们只要比较另外的一部分即两个端点，p[next[k]]和p[j](对应于代码中的p[k]==p[j]，注意在上个循环p[k]!=p[j]时，k已经被赋值next[k],而j还是上次的那个j);如果这两者相等了,则重复子串的长度+1，next[j+1]=next[k]+1(k++即next[k]+1);如果不相等了，则说明倒数第二大的p[0~next[k]-1]都不行了，比这个重复子串小的最大的重复子串只能是k=next[next[k]]了，如此继续查找下去。因此比较的都是按序递减的最大重复子串，非常的有效，一点都没有多比较。找不到的话，k会被赋值为-1。

这个算法神奇难解之处就在k=next[k]这一处的理解上，网上解析的非常之多，有的就是例证，举例子按代码走流程，走出结果了，跟肉眼看的一致，就认为解释了为什么k=next[k]；很少有看到解释的非常清楚的，或者有，但我没有仔细和耐心看下去。我一般扫一眼，就大概知道这个解析是否能说的通。仔细想了三天，搞的千转百折，山重水复，一头雾气缭绕的。搞懂以后又觉得确实简单，但是绕人，烧脑。

有了next数组之后就一切好办了，我们可以动手写KMP算法了：

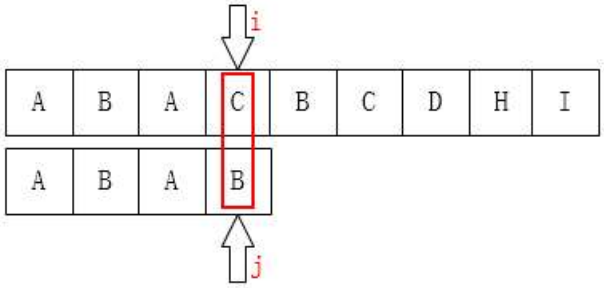
```
1 public static int KMP(String ts, String ps) {
2
3     char[] t = ts.toCharArray();
4
5     char[] p = ps.toCharArray();
6
7     int i = 0; // 主串的位置
8
9     int j = 0; // 模式串的位置
10
11    int[] next = getNext(ps);
12
13    while (i < t.length && j < p.length) {
14
15        if (j == -1 || t[i] == p[j]) { // 当j为-1时，要移动的是i，当然j也要归0
16
17            i++;
18
19            j++;
20
21        } else {
22
23            // i不需要回溯了
24
25            // i = i - j + 1;
26
27            j = next[j]; // j回到指定位置
28
29        }
30
31    }
32
33    if (j == p.length) {
34
35        return i - j;
```

```
36
37      } else {
38
39          return -1;
40
41      }
42
43  }
```

和暴力破解相比，就改动了4个地方。其中最主要的一点就是，i不需要回溯了。

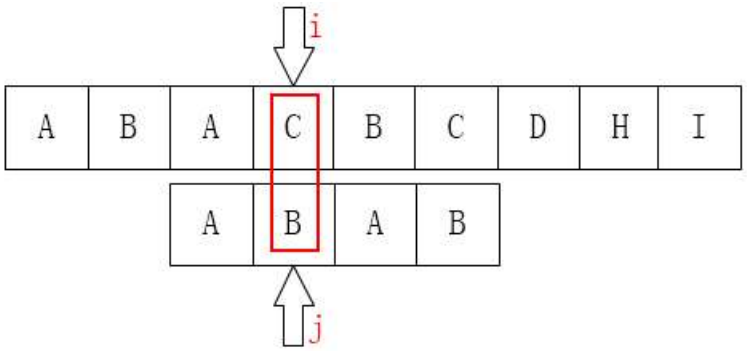
4.next数组求解算法优化

最后，来看一下上边的算法存在的缺陷。来看第一个例子：



显然，当我们上边的算法得到的next数组应该是[-1, 0, 0, 1]

所以下一步我们应该是把j移动到第1个元素咯：



不难发现，**这一步是完全没有意义的。因为后面的B已经不匹配了，那前面的B也一定是不匹配的**，同样的情况其实还发生在第2个元素A上。

显然，**发生问题的原因在于P[j] == P[next[j]]。**

补充说明：这部分作者说的也比较清楚了。实际上对下面的代码if(p[++j]==p[++k])，我们注意是先自加，再使用。所以我们按照j不变的情况下解释下一步流程就是p[j+1]==p[next[j]+1];此时比较将无意义，因为p[next[k]+1]位就已经表示，就是k+1位和主串的i不相等，要移动的j下标为next[K+1]，因为p[k+1]又等于p[j+1],也就是说比较j+1位和主串的i位是否相等时，也将要j移到next[K+1]位去；

所以我们也只需要添加一个判断条件即可：

```
public static int[] getNext(String ps) {

    char[] p = ps.toCharArray();

    int[] next = new int[p.length];

    next[0] = -1;

    int j = 0;

    int k = -1;
```

```
    while (j < p.length - 1) {
        if (k == -1 || p[j] == p[k]) {
            if (p[++j] == p[++k]) { // 当两个字符相等时要跳过
                next[j] = next[k];
            } else {
                next[j] = k;
            }
        } else {
            k = next[k];
        }
    }
    return next;
}
```

对我这个解释，有疑问的，欢迎探讨。

我十多年前刚工作的时候，实在没想到十多年后的不惑之年，居然重新开始做程序员，才感叹自己已经不写程序好久了，对自己是否还能写程序，还能有那个热情，产生了隐隐作痛的怀疑。为了新生活，为了在异国他乡重新找工作，我给自己定了目标，学ASP.NET、前端开发和算法。犹记读书时，老师的数据结构课程中对KMP算法，就略过不讲。我自己看的，我记得当时应该是看懂的。可现在我居然又想了两天，其中好几次，在懂了，又不懂了的过程中恍惚徘徊，我都怀疑是不是真的年纪大了，脑子不好使了。心情异常沮丧也无可奈何，自己的选择，必须坚定的走下去。本篇是我第一篇程序员博客，将纪录我的新的人生历程。我目前的主要关注点在ASP.NET MVC和REACT等前端开发，并且开始刷leetcode题目。希望自己能挺下去。

分类: algorithm

标签: 算法

sofu6
关注 - 9
粉丝 - 33
+加关注

34 1

» 下一篇: 全排列和康托展开(转载+注释)