**字符串匹配算法知识点**

Boyer-Moore算法、Horspool算法、Sunday算法、KMP算算法、RK算法、AC自动机

<http://dsqiu.iteye.com/blog/1700312>

<http://www.cnblogs.com/Franky-ln/p/5890201.html>

**一、KMP算法**

<http://www.cnblogs.com/c-cloud/p/3224788.html>

<http://jakeboxer.com/blog/2009/12/13/the-knuth-morris-pratt-algorithm-in-my-own-words/>

<http://www.61mon.com/index.php/archives/183/>

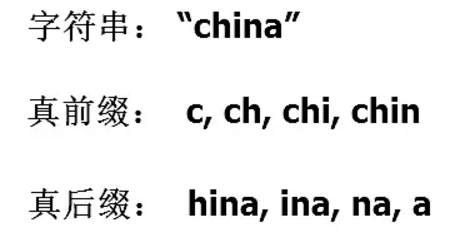
<http://www.cnblogs.com/SYCstudio/p/7194315.html>

<https://segmentfault.com/a/1190000007066358>

**1、背景介绍**

给定一个**主字符串（以 S 代替）和模式串（以 P 代替）**，要求找出 P 在 S 中出现的位置，即串的模式匹配问题。今天来介绍解决这一问题的常用算法之一，Knuth-Morris-Pratt 算法（简称 KMP），这个算法是由高德纳（Donald Ervin Knuth）和沃恩 · 普拉特在 1974 年构思，同年詹姆斯 ·H· 莫里斯也独立地设计出该算法，最终由三人于 1977 年联合发表。

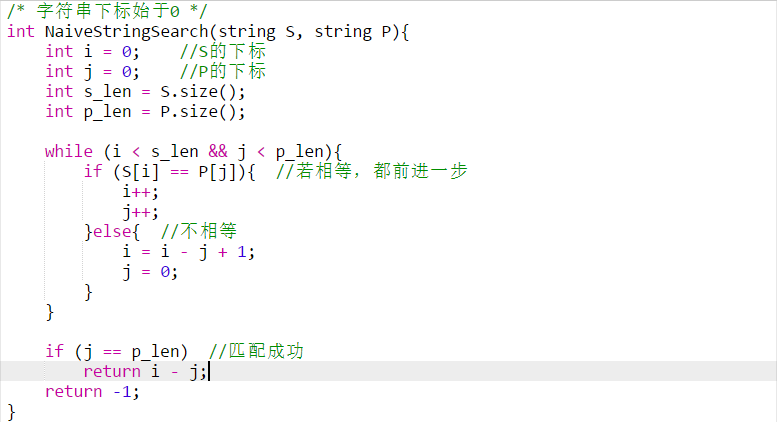
在继续下面的内容之前，有必要在这里介绍下两个概念：真前缀和真后缀。



由上图所得， **"真前缀" 指除了自身以外，一个字符串的全部头部组合；"真后缀" 指除了自身以外，一个字符串的全部尾部组合。**（网上很多博客，应该说是几乎所有的博客，也包括我以前写的，都是 “前缀”。严格来说，“真前缀” 和“前缀”是不同的，既然不同，还是不要混为一谈的好！）

**2、朴素字符串匹配算法**

初遇串的模式匹配问题，我们脑海中的第一反应，必是朴素字符串匹配（暴力匹配），即遍历 S 的每个字符，以该字符为始与 P 比较，全部匹配就输出；否则直到 S 结束。代码如下：（逐步比较）



上述算法的时间复杂度为 **O(nm)**，其中 n 为 S 的长度，m 为 P 的长度。这种时间复杂度很难满足我们的需求，接下来进入正题：时间复杂度为 **Θ(n+m) 的 KMP 算法**。

**3、KMP字符串匹配算法**

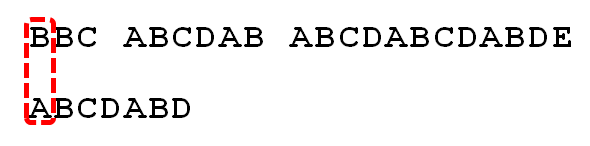
**1）理解1**

<http://www.61mon.com/index.php/archives/183/>

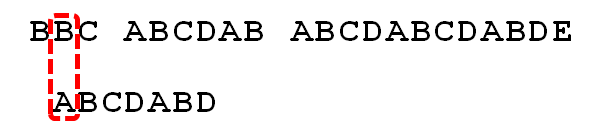
**A. 算法流程**

举例来说，有一个字符串"BBC ABCDAB ABCDABCDABDE"，我想知道，里面是否包含另一个字符串"ABCDABD"？

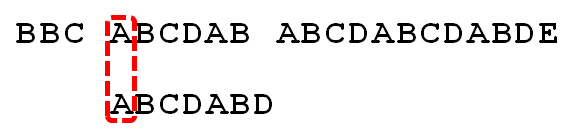
* 首先，字符串"BBC ABCDAB ABCDABCDABDE"的第一个字符与搜索词"ABCDABD"的第一个字符，进行比较。因为B与A不匹配，所以搜索词后移一位。



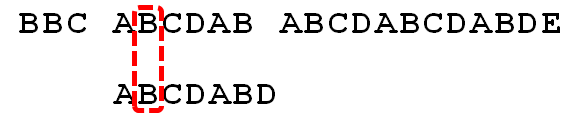
* 因为B与A不匹配，搜索词再往后移。



* 就这样，直到字符串有一个字符，与搜索词的第一个字符相同为止。



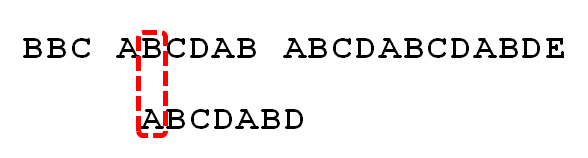
* 接着比较字符串和搜索词的下一个字符，还是相同。



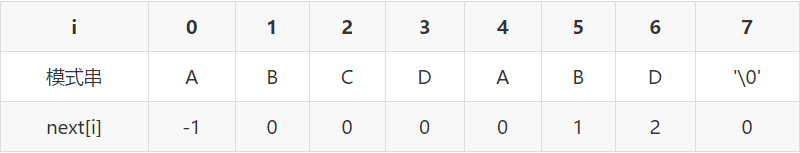
* 直到字符串有一个字符，与搜索词对应的字符不相同为止



* 这时，最自然的反应是，将搜索词整个后移一位，再从头逐个比较。这样做虽然可行，但是效率很差，因为你要把"搜索位置"移到已经比较过的位置，重比一遍。



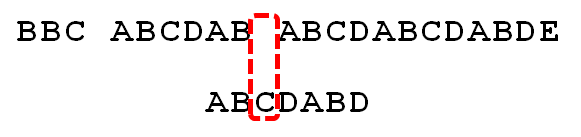
* 一个**基本事实**是，当空格与D不匹配时，你其实知道前面六个字符是"ABCDAB"。KMP算法的想法是，设法利用这个已知信息，***不要把"搜索位置"移回已经比较过的位置***，继续把它向后移，这样就提高了效率。
* 怎么做到这一点呢？可以针对搜索词，算出一张**《部分匹配表》**（Partial Match Table）。这张表是如何产生的，后面再介绍，这里只要会用就可以了。



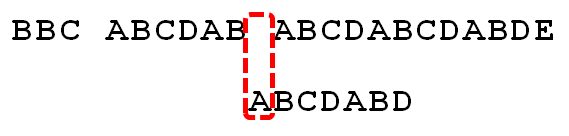
* 已知空格与 D 不匹配时，前面六个字符 "ABCDAB" 是匹配的。根据跳转数组可知，**不匹配处 D 的 next 值为 2**，因此接下来从模式串下标为 2 的位置开始匹配。



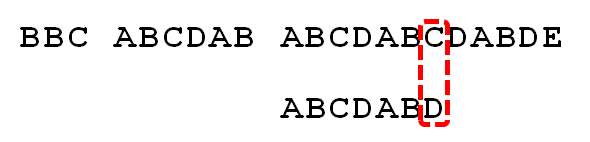
* 因为空格与Ｃ不匹配，C 处的 next 值为 0，因此接下来模式串从下标为 0 处开始匹配。



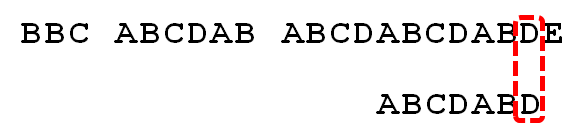
* 因为空格与 A 不匹配，此处 **next 值为 - 1**，表示模式串的第一个字符就不匹配，那么直接往后移一位



* 逐位比较，直到发现 C 与 D 不匹配。于是，下一步从下标为 2 的地方开始匹配。



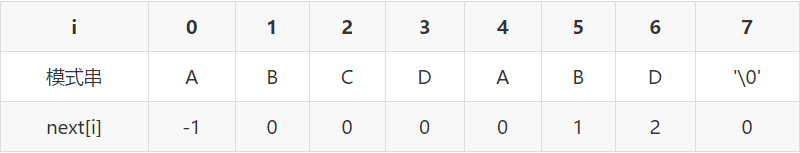
* 逐位比较，直到模式串的最后一位，发现完全匹配，于是搜索完成。



**B. next数组求解**

<http://www.61mon.com/index.php/archives/183/>

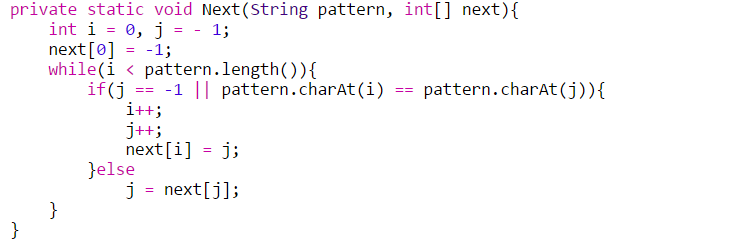
next 数组（部分匹配数组）的求解基于 “真前缀” 和“真后缀”，即*next[i]等于P[0]...P[i - 1]最长的相同真前后缀的长度*（请暂时忽视 i 等于 0 时的情况，下面会有解释）。我们依旧以上述的表格为例，为了方便阅读，我复制在下方了。



* i = 0，对于模式串的**首字符**，我们统一为next[0] = -1；
* i = 1，前面的字符串为A，其最长相同真前后缀长度为 0，即next[1] = 0；
* i = 2，前面的字符串为AB，其最长相同真前后缀长度为 0，即next[2] = 0；
* i = 3，前面的字符串为ABC，其最长相同真前后缀长度为 0，即next[3] = 0；
* i = 4，前面的字符串为ABCD，其最长相同真前后缀长度为 0，即next[4] = 0；
* i = 5，前面的字符串为ABCDA，其最长相同真前后缀为**A**，即next[5] = 1；
* i = 6，前面的字符串为ABCDAB，其最长相同真前后缀为**AB**，即next[6] = 2；
* i = 7，前面的字符串为ABCDABD，其最长相同真前后缀长度为 0，即next[7] = 0。

那么，为什么根据最长相同真前后缀的长度就可以实现在不匹配情况下的跳转呢？举个代表性的例子：假如i = 6时不匹配，此时我们是知道其位置前的字符串为ABCDAB，仔细观察这个字符串，**首尾都有一个AB**，既然在i = 6处的 D 不匹配，我们为何不直接把i = 2处的 C 拿过来继续比较呢，因为都有一个AB啊，而这个AB就是ABCDAB的最长相同真前后缀，其长度 2 正好是跳转的下标位置。

有的读者可能存在疑问，若在**i = 5时匹配失败**，按照我讲解的思路，此时应该把i = 1处的字符拿过来继续比较，但是这两个位置的字符是一样的啊，都是B，既然一样，拿过来比较不就是无用功了么？其实不是我讲解的有问题，也不是这个算法有问题，而是这个算法还未优化，关于这个问题在下面会详细说明，不过建议读者不要这里纠结，跳过这个，下面你自然会恍然大悟。



上述代码是求解每个位置的 next 值，即***求解每个位置前面字符串的最长相同真前后缀的长度***。下面具体分析，我把代码分为 3 部分来讲：

* **i 的作用是什么？**

i 为模式串 P 的下标，从 0 开始，程序中我们依次求出next[i]的值，这很简单。

* **j 的作用是什么？**

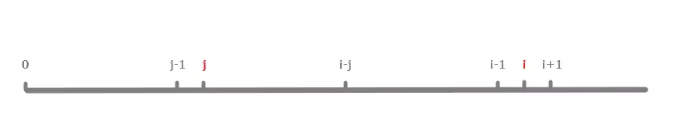
其一，从 if 语句中P[i] == P[j]，它作为模式串的下标，*判断下一个字符是否相等*。其二，从next[i] = j；可以很容易推断出，**j 代表最长相同真前后缀的长度**。两者在数值上正好相等。（之所以正好相等，是因为 **i 和 j 初始化**的时候，正好差 1，这点读者仔细思考下就能明白的）

* **if...else... 语句里做了什么？**

首先我们必须要明确一个事实：若此时i = 3，那我们接下来要求解的便是**P[0]...p[3]**的*最长相同真前后缀的长度*，也就是next[4]，而非next[3]，这从下面的代码就可以得到证明：

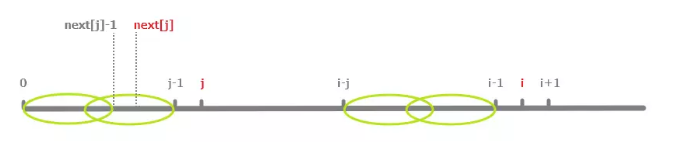


下面具体分析：



假设 i 和 j 的位置如上图，由**next[i] = j**得，也就是对于位置 i 来说，区段 0 到 i - 1 的最长相同真前后缀分别是 **0 到 j - 1** 和 **i - j 到 i - 1**，即这两区段内容相同。

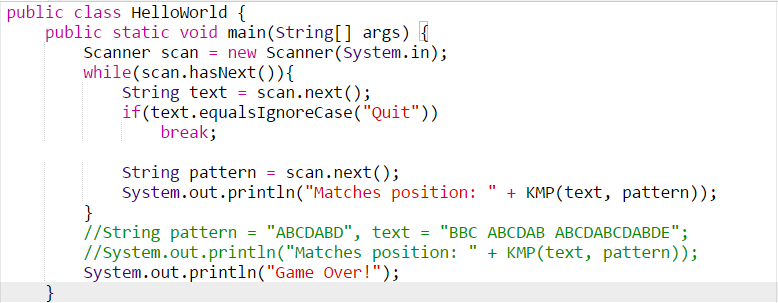
按照算法流程，if (P[i] == P[j])，则i++; j++; next[i] = j;；若不等，则j = next[j]，见下图：

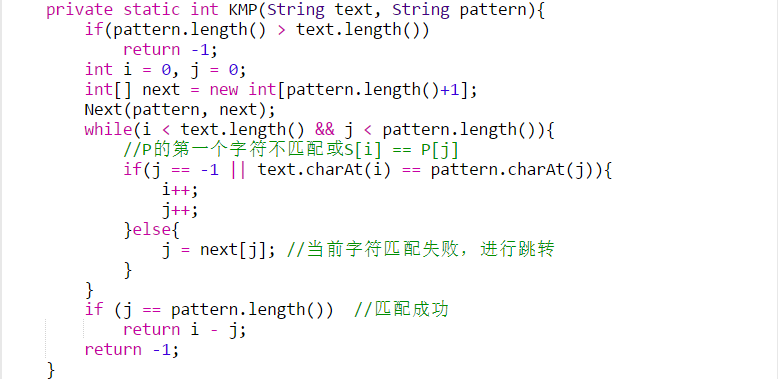


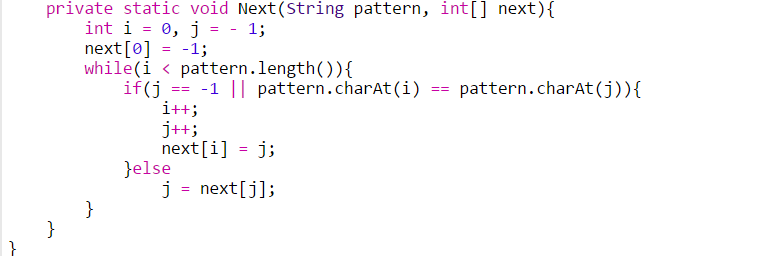
**next[j]代表 0 到 j - 1 区段中最长相同真前后缀的长度**。如图，用左侧两个椭圆来表示这个最长相同真前后缀，即这两个椭圆代表的区段内容相同；同理，右侧也有相同的两个椭圆。所以 else 语句就是利用**第一个椭圆和第四个椭圆内容相同**来加快得到 0 到 i - 1 区段的相同真前后缀的长度。

细心的朋友会问 if 语句中j == -1存在的意义是何？第一，程序刚运行时，j 是被初始为 - 1，直接进行P[i] == P[j]判断无疑会边界溢出；第二，else 语句中j = next[j]，***j 是不断后退的，若 j 在后退中被赋值为 - 1***（也就是j = next[0]），在P[i] == P[j]判断也会边界溢出。综上两点，其意义就是为了**特殊边界判断**。

**C. 完整代码**



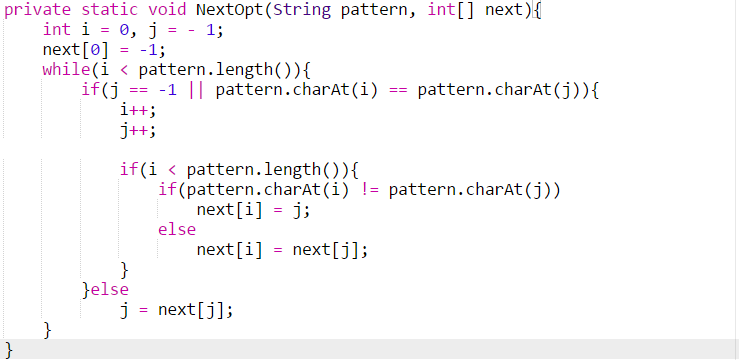




**D. KMP优化**



以 3.2 的表格为例（已复制在上方），若在i = 5时匹配失败，按照 3.2 的代码，此时应该**把i = 1处的字符拿过来继续比较**，但是这两个位置的字符是一样的，都是B，既然一样，拿过来比较不就是无用功了么？这我在 3.2 已经解释过，之所以会这样是因为 KMP 不够完美。那怎么改写代码就可以解决这个问题呢？很简单。



在此也给各位读者提个醒，KMP 算法严格来说分为 KMP 算法（未优化版）和 KMP 算法（优化版），所以建议读者在表述 KMP 算法时，最好告知你的版本，因为两者在某些情况下区别很大，这里简单说下。

* **KMP 算法（未优化版）：** next 数组表示***最长的相同真前后缀的长度***，我们不仅可以利用 next 来解决模式串的匹配问题，也可以用来解决**类似字符串重复问题**等等，这类问题大家可以在各大 OJ 找到，这里不作过多表述。
* **KMP 算法（优化版）：** 根据代码很容易知道（名称也改为了 nextval），优化后的 next 仅仅***表示相同真前后缀的长度***，但不一定是最长（我个人称之为 “最优相同真前后缀”）。此时我们利用优化后的 next 可以在模式串匹配问题中以更快的速度得到我们的答案（相较于未优化版），但是上述所说的字符串重复问题，优化版本则束手无策。

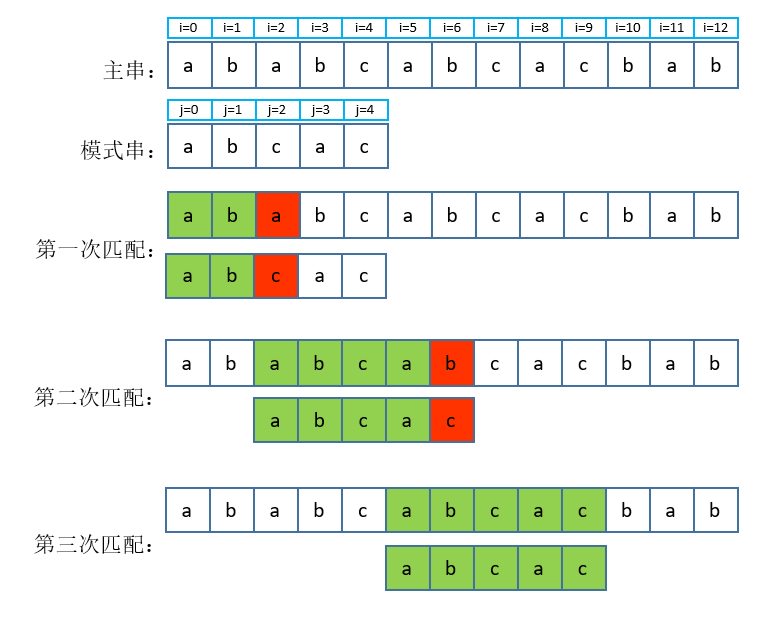
所以，该采用哪个版本，取决于你在现实中遇到的实际问题

**2）理解2**

**A. 算法流程及next数组求解**

<https://segmentfault.com/a/1190000007066358#articleHeader2>

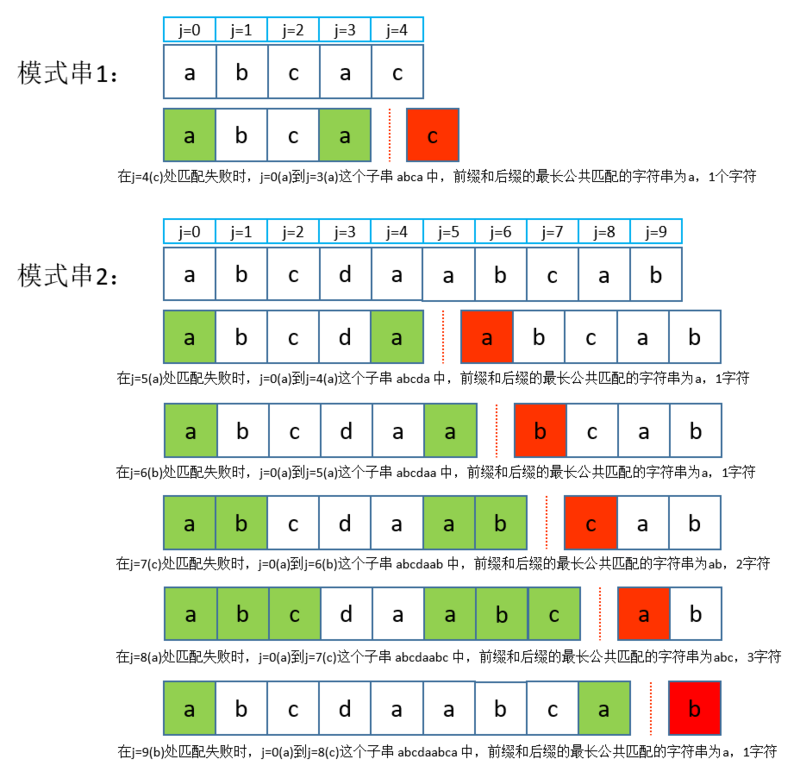
KMP算法匹配过程：



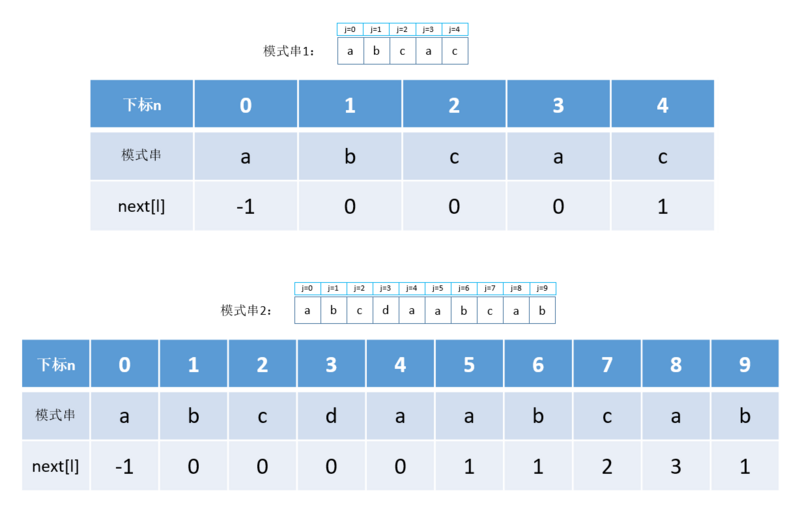
KMP算法的改进之处在于：**能够知道在匹配失败后，有多少字符是不需要进行匹配可以直接跳过的**，匹配失败后，下一次匹配从什么地方开始能够有效的减少不必要的匹配过程。

**next[n]求解方法：**

由上面的分析可以发现，KMP算法的核心在于对**模式串本身的分析**，其分析结果能*提供在j = n位置匹配失败时，从j = 0到j = n - 1这个子串中前缀和后缀的最长公共匹配的字符数*，这样说可能比较难以理解，看下图：



在得到**子串前缀和后缀的最长公共匹配字符数**l后，以后在**i = x,j = n处匹配失败时，可以直接从i = x,j = l处继续匹配**(证明过程参考:严蔚敏的《数据结构》4.3章)，这样问题就很明显了，我们要求出n和l对应的值，其中***n是模式串字符数组的下标***，l的有序集合通常称之为next数组，前面两个模式串的next数组和下标n的对应如下：



**模式串2完整匹配过程：**

有了这个next数组，那么在匹配的过程中我们就能在j = n处匹配失败后，根据next[n]的值进行**偏移**，其中***next[0]固定为-1，代表在当前i这个位置整个模式串和主串都无法匹配成功***，要从***下一个位置i = i + 1及j = 0处开始匹配***，模式串2的匹配过程如下：



现在知道了next数组的作用，也知道在有next数组时的匹配过程，那么剩下的问题就是如何通过代码求出next数组及匹配过程了。

**求next数组的过程可以认为是将模式串拆分成n个子串，分别对每个子串求前缀和后缀的最长公共匹配字符数l，这一点可以通过上图(最长公共匹配字符数)看出来(没有画出l=0时的图解)看出来。**

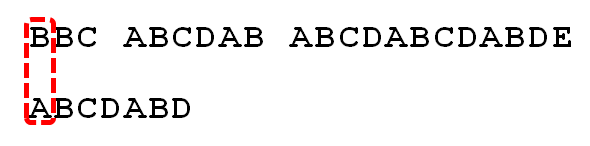
**3）理解3**

**A. 算法流程**

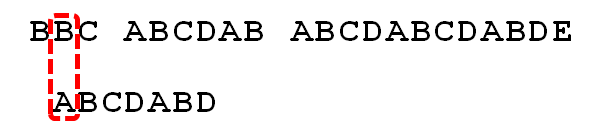
<http://www.cnblogs.com/c-cloud/p/3224788.html>

举例来说，有一个字符串"BBC ABCDAB ABCDABCDABDE"，我想知道，里面是否包含另一个字符串"ABCDABD"？

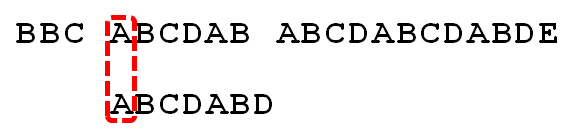
* 首先，字符串"BBC ABCDAB ABCDABCDABDE"的第一个字符与**搜索词**"ABCDABD"的第一个字符，进行比较。因为B与A不匹配，所以搜索词后移一位。



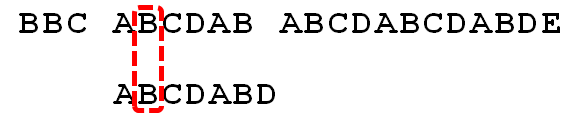
* 因为B与A不匹配，搜索词再往后移。



* 就这样，直到字符串有一个字符，与搜索词的第一个字符相同为止。



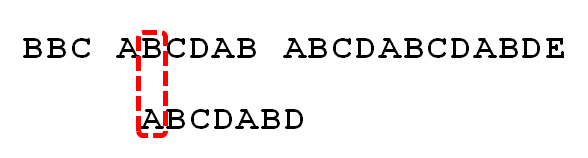
* 接着比较字符串和搜索词的下一个字符，还是相同。



* 直到字符串有一个字符，与搜索词对应的字符不相同为止



* 这时，最自然的反应是，将搜索词整个后移一位，再从头逐个比较。这样做虽然可行，但是效率很差，因为你要把"搜索位置"移到已经比较过的位置，重比一遍。



* 一个**基本事实**是，当空格与D不匹配时，你其实知道前面六个字符是"ABCDAB"。KMP算法的想法是，设法利用这个已知信息，***不要把"搜索位置"移回已经比较过的位置***，继续把它向后移，这样就提高了效率。
* 怎么做到这一点呢？可以针对搜索词，算出一张**《部分匹配表》**（Partial Match Table）。这张表是如何产生的，后面再介绍，这里只要会用就可以了。

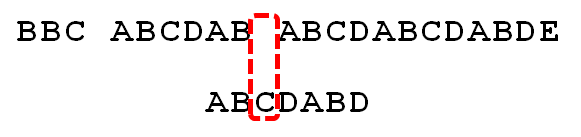


* 已知空格与D不匹配时，前面六个字符"ABCDAB"是匹配的。查表可知，*最后一个匹配字符B对应的"部分匹配值"为2*，因此按照下面的公式算出向后移动的位数：

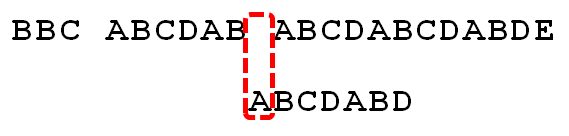
**移动位数 = 已匹配的字符数 - 对应的部分匹配值（最后一位匹配字符）**

因为 6 - 2 等于4，所以将搜索词向后移动4位。

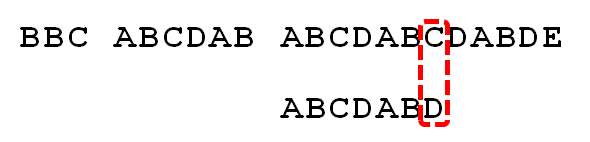
* 因为空格与Ｃ不匹配，搜索词还要继续往后移。这时，已匹配的字符数为2（"AB"），对应的**"部分匹配值"**为0。所以，移动位数 = 2 - 0，结果为 2，于是将搜索词向后移2位。



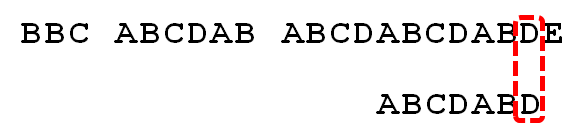
* 因为空格与A不匹配，继续后移一位



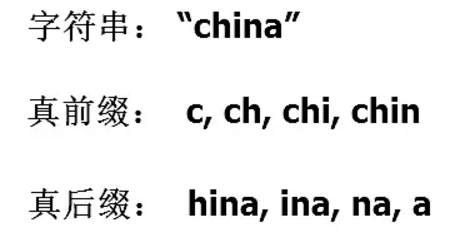
* 逐位比较，直到***发现C与D不匹配***。于是，移动位数 = 6 - 2，继续将搜索词向后移动4位。



* 逐位比较，直到搜索词的最后一位，发现完全匹配，于是搜索完成。如果还要继续搜索（即**找出全部匹配**），移动位数 = 7 - 0，再将搜索词向后移动7位，这里就不再重复了。



* 下面介绍**《部分匹配表》**是如何产生的。首先，要了解两个概念：**"真前缀"和"真后缀"**。 "真前缀"指*除了最后一个字符以外，一个字符串的全部头部组合*；"真后缀"指*除了第一个字符以外，一个字符串的全部尾部组合*。

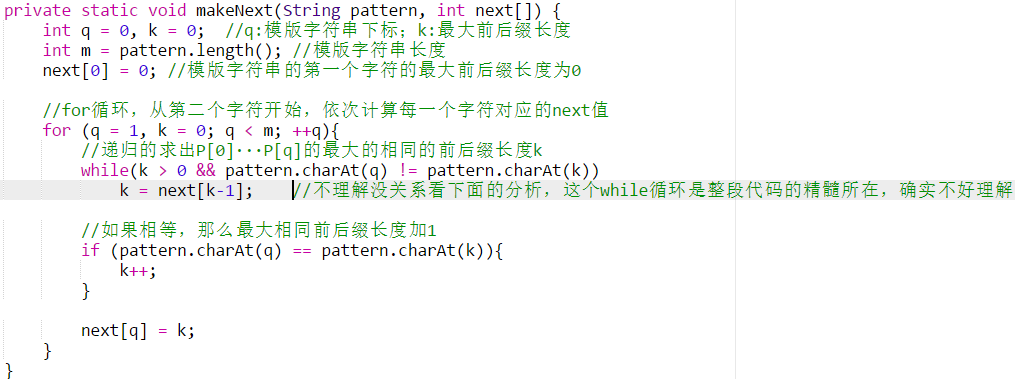


* "部分匹配值"就是**"前缀"和"后缀"的最长的共有元素的长度**。以"ABCDABD"为例，
* "A"的前缀和后缀都为空集，共有元素的长度为0；
* "AB"的前缀为[A]，后缀为[B]，共有元素的长度为0；
* "ABC"的前缀为[A, AB]，后缀为[BC, C]，共有元素的长度0；
* "ABCD"的前缀为[A, AB, ABC]，后缀为[BCD, CD, D]，共有元素的长度为0；
* "ABCDA"的**前缀**为[**A**, AB, ABC, ABCD]，**后缀**为[BCDA, CDA, DA, **A**]，共有元素为"A"，长度为1；
* "ABCDAB"的**前缀**为[A, **AB**, ABC, ABCD, ABCDA]，**后缀**为[BCDAB, CDAB, DAB, **AB**, B]，共有元素为"AB"，长度为2；
* "ABCDABD"的**前缀**为[A, AB, ABC, ABCD, ABCDA, ABCDAB]，**后缀**为[BCDABD, CDABD, DABD, ABD, BD, D]，共有元素的长度为0。

**"部分匹配"的实质**是，有时候，*字符串头部和尾部会有重复*。比如，"**AB**CD**AB**"之中有两个"AB"，那么它的**"部分匹配值"就是2（"AB"的长度）**。搜索词移动的时候，第一个"AB"向后移动4位（字符串长度-部分匹配值），就可以来到第二个"AB"的位置。

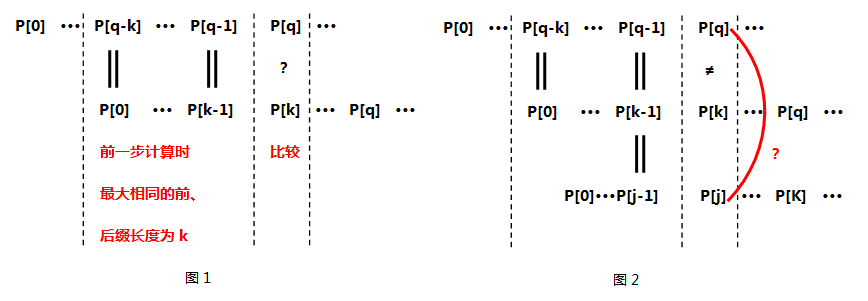
**B. next数组求解**

如何根据待匹配的模版字符串求出对应每一位的**最大相同前后缀的长度：**

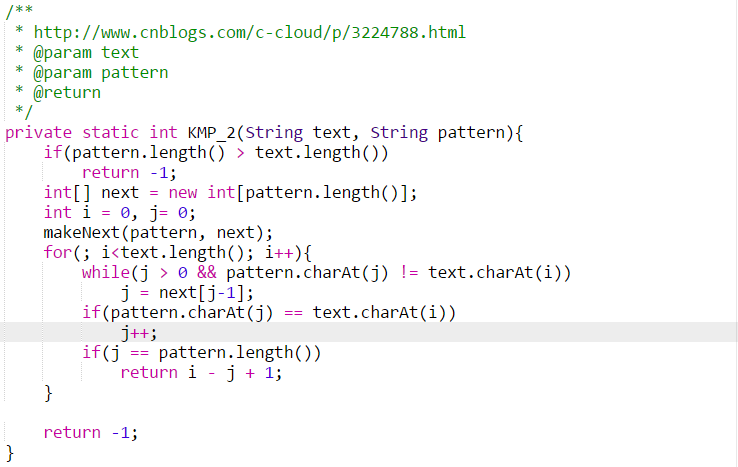


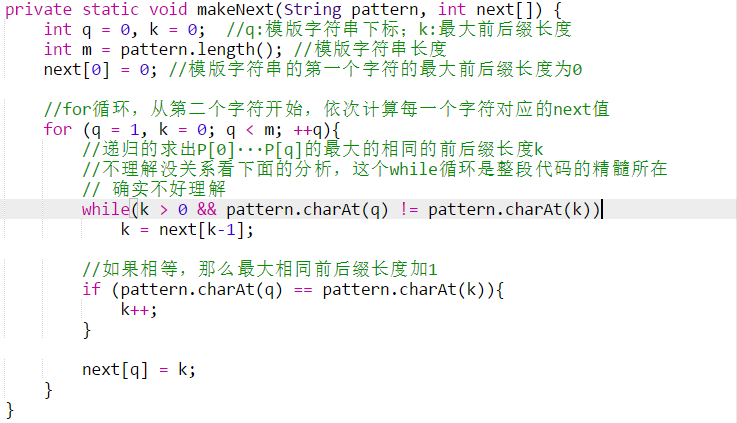
现在我着重讲解一下while循环所做的工作：

* 已知前一步计算时**最大相同的前后缀长度**为k（k>0），即**P[0]···P[k-1]**；
* 此时比较第k项P[k]与P[q],如图1所示
* 如果**P[K]等于P[q]**，那么很简单跳出while循环;
* 关键！关键有木有！关键如果不等呢？？？那么我们应该利用已经得到的next[0]···next[k-1]来求P[0]···P[k-1]这个**子串中最大相同前后缀**，可能有同学要问了——为什么要求P[0]···P[k-1]的最大相同前后缀呢？？？是啊！为什么呢？ 原因在于P[k]已经和P[q]失配了，而且**P[q-k] ··· P[q-1]**又与**P[0] ···P[k-1]**相同，看来P[0]···P[k-1]这么长的子串是用不了了，那么我要找个同样也是**P[0]打头、P[k-1]结尾**的子串即P[0]···P[j-1](j==next[k-1])，看看它的下一项P[j]是否能和P[q]匹配。如图2所示



**C. 完整代码**





**二、Boyer-Moore算法**

<http://blog.jobbole.com/52830/>

在用于查找子字符串的算法当中，**BM（Boyer-Moore）**算法是目前被认为***最高效的字符串搜索算法***，它由Bob Boyer和J Strother Moore设计于1977年。 一般情况下，比KMP算法快3-5倍。该算法**常用于文本编辑器中的搜索匹配功能**，比如大家所熟知的GNU grep命令使用的就是该算法，这也是GNU grep比BSD grep快的一个重要原因。

GNU grep使用了非常著名的Boyer-Moore算法，该算法**首先**从目标字符串的最后一个字符开始查找，并且使用一个**查找表**，它可以在发现一个不匹配字符之后，计算出可以跳过多少个输入字符并继续查找。

**1、主要特征**

假设文本串text长度为n，模式串pattern长度为m，BM算法的主要特征为：

从右往左进行比较匹配（一般的字符串搜索算法如KMP都是从**从左往右进行匹配**）；

* 算法分为两个阶段：预处理阶段和搜索阶段；
* 预处理阶段时间和空间复杂度都是是O(m+sigma)，sigma是字符集大小，一般为256；
* 搜索阶段时间复杂度是O(mn)；
* 当**模式串**是非周期性的，在**最坏**的情况下算法需要进行3n次字符比较操作；
* 算法在最好的情况下达到O(n / m)，比如在文本串bn中搜索模式串am-1b ，只需要n/m次比较。

**2、算法基本思想**

常规的匹配算法移动模式串的时候是**从左到右**，而进行比较的时候也是从左到右的，基本框架是：



而BM算法在移动模式串的时候是**从左到右**，而进行比较的时候是**从右到左的**，基本框架是：



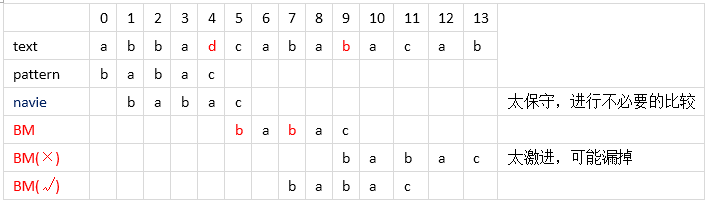
BM算法的精华就在于**BM(text, pattern)**，也就是*BM算法当不匹配的时候一次性可以跳过不止一个字符*。即它不需要对被搜索的字符串中的字符进行逐一比较，而会跳过其中某些部分。通常**搜索关键字越长，算法速度越快**。它的效率来自于这样的事实：对于每一次失败的匹配尝试，算法都能够使用这些信息来排除尽可能多的无法匹配的位置。即它充分利用待搜索字符串的一些特征，加快了搜索的步骤。

BM算法实际上包含两个**并行的算法**（也就是两个启发策略）：坏字符算法（bad-character shift）和好后缀算法（good-suffix shift）。这两种算法的目的就是*让模式串每次向右移动尽可能大的距离（即上面的BM()尽可能大）*。

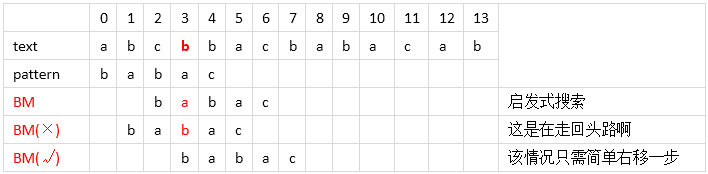
下面不直接书面解释这两个算法，为了更加通俗易懂，先用实例说明吧，这是最容易接受的方式。

**3、实例解析**

如何加快字符串搜索？举个很简单的例子，如下图所示，navie表示一般做法，逐个进行比对，**从右向左**，最后一个字符c与text中的d不匹配，pattern右移一位。但大家看一下这个d有什么特征？pattern中没有d，因此你不管右移1、2、3、4位肯定还是不匹配，何必花这个功夫呢？**直接右移5（strlen(pattern)）位**再进行比对不是更好吗？好，就这样做，右移5位后，text中的b与pattern中的c比较，发现还是不同，这时咋办？b在pattern中有所以不能一下右移5位了，难道直接右移一位吗？No，可以***直接将pattern中的b右移到text中b的位置进行比对***，但是pattern中有两个b，右移哪个b呢？保险的办法是用最右边的b与text进行比对，为啥？下图说的很清楚了，用最左边的b太激进了，容易漏掉真正的匹配，图中用最右边的b后发现正好所有的都匹配成功了，如果用最左边的不就错过了这个匹配项吗？这个启发式搜索就是BM算法做的。

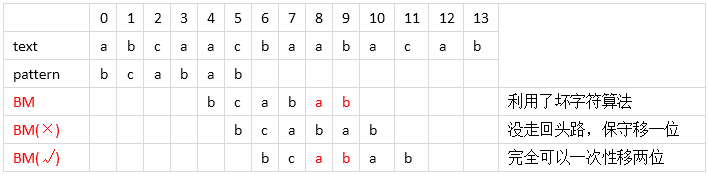


But, 如果遇到下面这样的情况，开始pattern中的c和text中的b不匹配，Ok，按上面的规则*将pattern右移直至最右边的b与text的b对齐进行比对*。再将pattern中的c与text中的c进行比对，匹配继续**往左比对**，直到位置3处pattern中的a与text中的b不匹配了，按上面讲的启发式规则应该将pattern中最右边的b与text的b对齐，可这时发现啥了？pattern走了回头路，干吗？当然不干，才不要那么傻，针对这种情况，只需要将pattern简单的右移一步即可，坚持不走回头路！

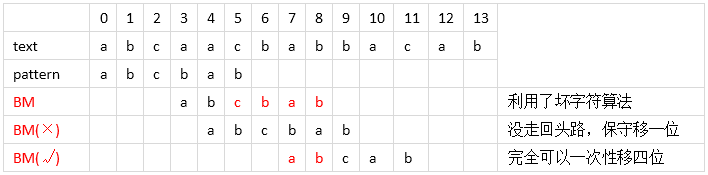


好了，这就是所谓的**“坏字符算法”**，简单吧，通俗易懂吧，上面用红色粗体字标注出来的b就是“坏字符”，即不匹配的字符，**坏字符是针对text的**。

BM难道就这么简单？就一个启发式规则就搞定了？当然不是了，大家再次头脑风暴一下，有没有其他加快字符串搜索的方法呢？比如下面的例子。



一开始利用了坏字符算法一下移了4位，不错，接下来遇到了回头路，没办法只能保守移一位，但真的就只能移一位吗？No，因为***pattern中前面其他位置也有刚刚匹配成功的后缀ab，那么将pattern前面的ab右移到text刚匹配成功的ab对齐继续往前匹配不是更好吗？***这样就可以一次性右移两位了，很好的有一个启发式搜索规则啊。有人可能想：要是前面没已经匹配成功的后缀咋办？是不是就无效了？不完全是，这要看情况了，比如下面这个例子。



cbab这个后缀已经成功匹配，然后b没成功，而pattern前面也没发现cbab这样的串，这样就直接保守移一位？No，前面有ab啊，这是cbab后缀的一部分，也可以好好利用，直接将pattern前面的ab右移到text已经匹配成功的ab位置处继续往前匹配，这样一下子就右移了四位，很好。当然，**如果前面完全没已经匹配成功的后缀或部分后缀**，比如最前面的babac，那就真的不能利用了。

好了，这就是所谓的**“好后缀算法”**，简单吧，通俗易懂吧，上面用红色字标注出来的ab（前面例子）和cbab（上面例子）就是“好后缀”，好后缀是针对pattern的。

下面，最后再举个例子说明啥是坏字符，啥是好后缀。

**主串：**mahtavaa**t**alomaisema omalomailuun

**模式串：**maisemao**m**aloma

**坏字符：**主串中的“t”为坏字符。

**好后缀：**模式串中的aloma为“好后缀”。

BM就这么简单？是的，容易理解但并不是每个人都能想到的两个启发式搜索规则就造就了BM这样一个优秀的算法。那么又有个问题？这两个算法怎么运用，一下坏字符的，一下好后缀的，**什么时候该用坏字符？什么时候该用好后缀呢？**很好的问题，这就要看哪个右移的位数多了，比如上面的例子，一开始如果用好后缀的话只能移一位而用坏字符就能右移三位，此时当然选择坏字符算法了。接下来如果继续用坏字符则只能右移一位而用好后缀就能一下右移四位，这时候你说用啥呢？So，这两个算法是“并行”的，哪个大用哪个。

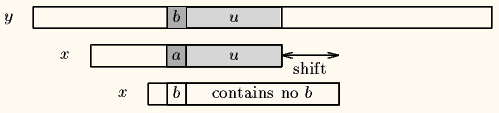
光用例子说明当然不够，太浅了，而且还不一定能完全覆盖所有情况，不精确。下面就开始真正的理论探讨了。

**4、BM算法理论探讨**

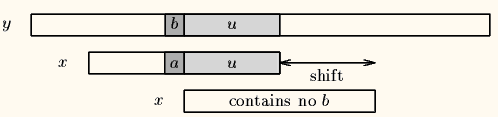
**1）坏字符算法：针对主串**

当出现一个坏字符时, **BM算法向右移动模式串**, 让*模式串中最靠右的对应字符与坏字符相对，然后继续匹配*。坏字符算法有两种情况。

* **Case1：**模式串中有对应的坏字符时，让模式串中最靠右的对应字符与坏字符相对（PS：BM不可能走回头路，因为若是回头路，则移动距离就是负数了，肯定不是最大移动步数了），如下图



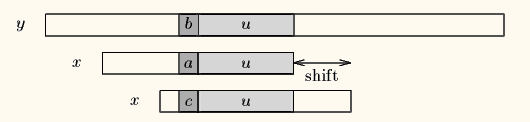
* **Case2：**模式串中不存在坏字符，很好，直接右移整个模式串长度这么大步数，如下图



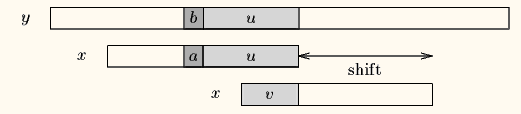
**2）好后缀算法**

如果程序匹配了一个**好后缀**, 并且*在模式中还有另外一个相同的后缀或后缀的部分, 那把下一个后缀或部分移动到当前后缀位置*。假如说，pattern的后u个字符和text都已经匹配了，但是接下来的一个字符不匹配，需要移动才能匹配。如果说后u个字符在pattern其他位置也出现过或部分出现，我们将**pattern右移**到前面的u个字符或部分和最后的u个字符或部分相同，如果说后u个字符在pattern其他位置完全没有出现，很好，直接右移整个pattern。这样，好后缀算法有三种情况，如下图所示：

* **Case1：**模式串中有子串和好后缀**完全匹配**，则将最靠右的那个子串移动到好后缀的位置继续进行匹配。



* **Case2：**如果不存在和好后缀完全匹配的子串，则在好后缀中找到具有如下特征的最长子串,使得**P[m-s…m]=P[0…s]**。



* **Case3：**如果完全不存在和好后缀匹配的子串，则右移整个模式串。

**3）移动规则**

**BM算法的移动规则**是：将3中算法基本框架中的**j += BM()**，换成j += MAX（shift（好后缀），shift（坏字符）），即

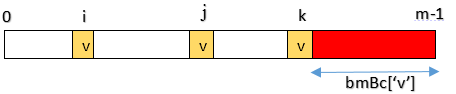
BM算法每次向右移动模式串的距离是，按照好后缀算法和坏字符算法计算得到的最大值。**shift（好后缀）和shift（坏字符）**通过模式串的预处理数组的简单计算得到。坏字符算法的预处理数组是**bmBc[]**，好后缀算法的预处理数组是**bmGs[]**。

**5、BM算法具体执行**

BM算法子串比较失配时，按坏字符算法计算pattern需要右移的距离，要借助bmBc数组，而按好后缀算法计算pattern右移的距离则要借助**bmGs**数组。下面讲下怎么计算bmBc[]和**bmGs**[]这两个预处理数组。**文本串text长度为n，模式串pattern长度为m**。

**1）计算坏字符数组bmBc[]**

这个计算应该很容易，似乎只需要**bmBc[i] = m–1–i**就行了，但这样是不对的，因为**i位置处的字符**可能在pattern中多处出现（如下图所示），而我们需要的是**最右边的位置**，这样就需要每次循环判断了，非常麻烦，性能差。这里有个小技巧，就是**使用字符作为下标**而不是位置数字作为下标。这样只需要遍历一遍即可，这貌似是空间换时间的做法，但如果是纯8位字符也只需要256个空间大小，而且对于大模式，可能本身长度就超过了256，所以这样做是值得的（这也是为什么数据越大，BM算法越高效的原因之一）。



如前所述，bmBc[]的计算分两种情况，与前一一对应。

**Case1：**字符在模式串中有出现，bmBc[‘v’]表示*字符v在****模式串****中最后一次出现的位置*，**距离模式串串尾的长度**，如上图所示。

**Case2：**字符在模式串中没有出现，如模式串中没有字符v，则BmBc[‘v’] = strlen(pattern)。

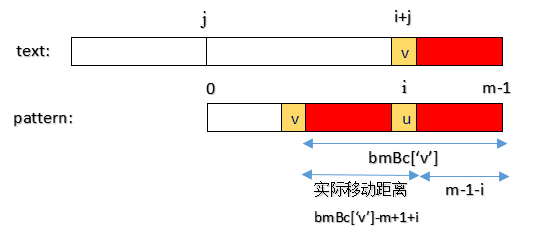
写成代码也非常简单：



或者



计算*pattern需要右移的距离*，要借助bmBc数组，那么bmBc的值是不是就是pattern实际要右移的距离呢？No，想想也不是，比如前面举例说到利用bmBc算法还可能走回头路，也就是右移的距离是**负数**，而**bmBc的值绝对不可能是负数**，所以两者不相等。那么pattern实际右移的距离怎么算呢？这个就要看text中坏字符的位置了，前面说过坏字符算法是针对text的，还是看图吧，一目了然。图中v是text中的坏字符（对应位置**i+j**）,在pattern中对应不匹配的位置为i，那么pattern实际要**右移的距离**就是：**bmBc[‘v’] – m + 1 + i**。

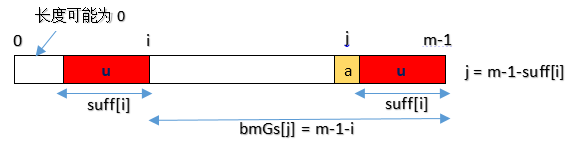


**2）计算好后缀数组bmGs[]**

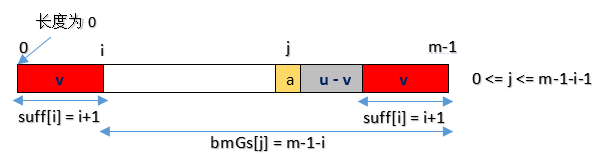
这里bmGs[]的***下标是数字而不是字符了，表示字符在pattern中位置***。

如前所述，bmGs数组的计算分三种情况，与前一一对应。假设图中好后缀长度用数组suff[]表示。

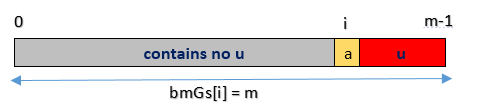
* **Case1：**对应好后缀算法case1，如下图，j是好后缀之前的那个位置。



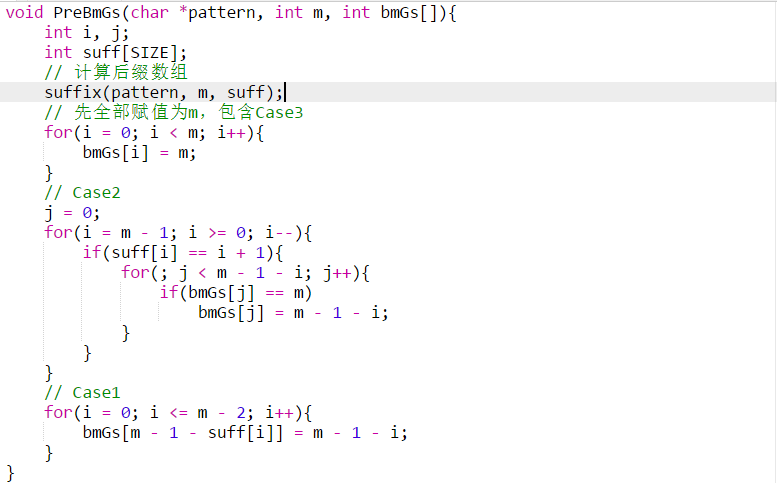
* **Case2：**对应好后缀算法case2：如下图所示：



* **Case3：**对应与好后缀算法case3，bmGs[i] = strlen（pattern）= m



这样就更加清晰了，代码编写也比较简单：



在计算bmGc数组时，为提高效率，先计算**辅助数组suff[]表示好后缀的长度**。

suff数组的定义：m是pattern的长度

a. suffix[m-1] = m;

b. suffix[i] = k

for [ **pattern[i-k+1] ….,pattern[i]] == [pattern[m-1-k+1]，pattern[m-1]**]

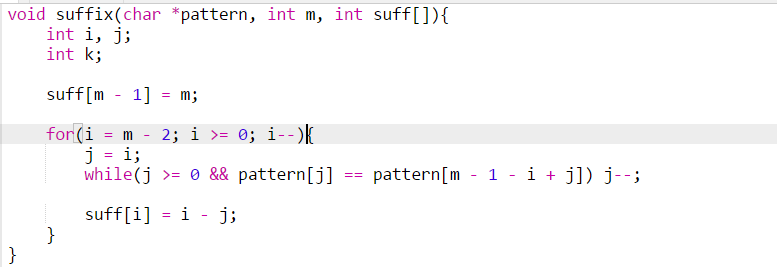
看上去有些晦涩难懂，实际上***suff[i]就是求pattern中以i位置字符为后缀和以最后一个字符为后缀的公共后缀串的长度***。不知道这样说清楚了没有，还是举个例子吧：

i : 0 1 2 3 4 5 6 7

pattern: b c a b a b a b

* 当i=7时，按定义suff[7] = strlen(pattern) = 8
* 当i=6时，以pattern[6]为后缀的**后缀串**为bcababa，以最后一个字符b为后缀的后缀串为bcababab，两者没有**公共后缀串**，所以suff[6] = 0
* 当i=5时，以pattern[5]为后缀的后缀串为bc**abab**，以最后一个字符b为后缀的后缀串为bcab**abab**，两者的公共后缀串为abab，所以suff[5] = 4
* 以此类推……
* 当i=0时，以pattern[0]为后缀的后缀串为**b**，以最后一个字符b为后缀的后缀串为bcababa**b**，两者的公共后缀串为b，所以suff[0] = 1。

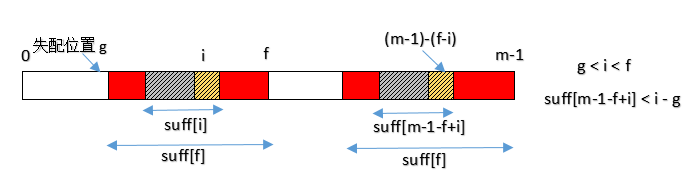
这样看来代码也很好写：



这样可能就万事大吉了，可是总有人对这个算法不满意，感觉太暴力了，于是有聪明人想出一种方法，对上述常规方法进行改进。基本的扫描都是**从右向左**，改进的地方就是*利用了已经计算得到的suff[]值，计算现在正在计算的suff[]值*。具体怎么利用，看下图：

* i是当前正准备计算suff[]值的那个位置。
* f是上一个成功进行匹配的起始位置（不是每个位置都能进行成功匹配的， 实际上能够进行成功匹配的位置并不多）。
* g是上一次进行成功匹配的失配位置。

如果i在g和f之间，那么一定有P[i]=P[m-1-f+i]；并且如果suff[m-1-f+i] < i-g, 则suff[i] = suff[m-1-f+i]，这不就利用了前面的suff了吗。



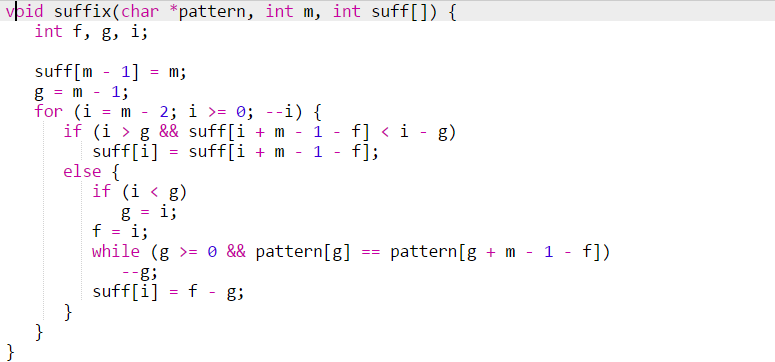
PS：这里有些人可能觉得应该是suff[m-1-f+i] <= i – g，因为若suff[m-1-f+i] = i – g，还是没超过suff[f]的范围，依然可以利用前面的suff[]，但这是错误的，比如一个极端的例子：

i ：0 1 2 3 4 5 6 7 8 9

pattern：a a a a a b a a a a

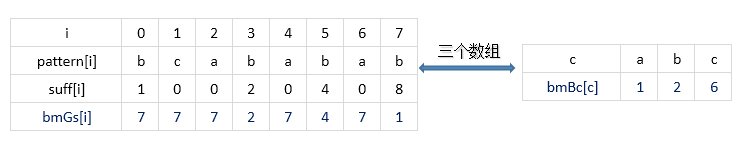
suff[4] = 4，这里f=4，g=0，当i=3是，这时suff[m-1=f+i]=suff[8]=3，而suff[3]=4，两者不相等，因为上一次的失配位置g可能会在这次得到匹配。

好了，这样解释过后，代码也比较简单：

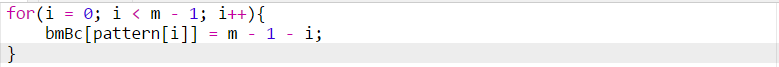


结束了？OK，可以说重要的算法都完成了，希望大家能够看懂，为了验证大家到底有没有完全看明白，下面出个简单的例子，大家算一下bmBc[]、suff[]和bmGs[]吧。

举例如下：



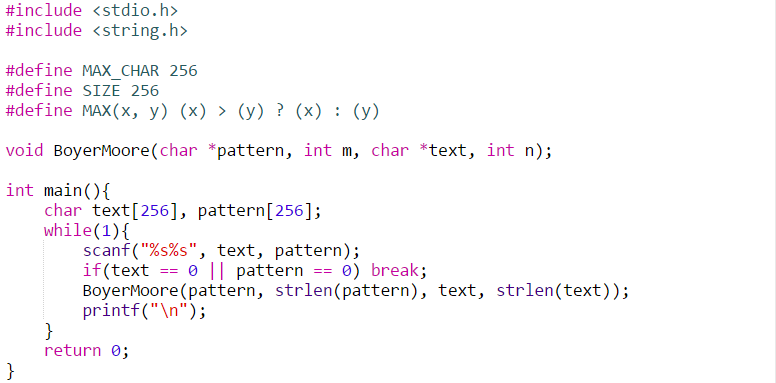
**PS：**这里也许有人会问：bmBc[‘b’]怎么等于2，它不是最后出现在pattern最后一个位置吗？按定义应该是0啊。请大家仔细看下bmBc的算法：

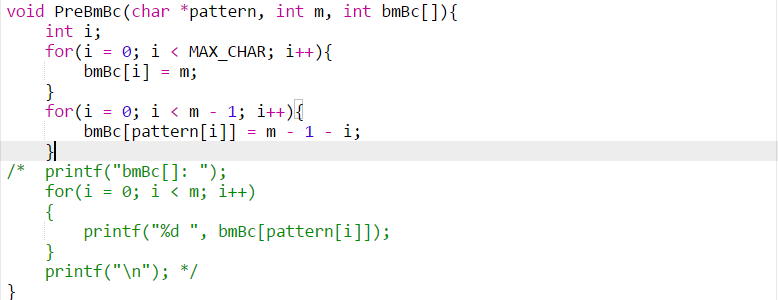


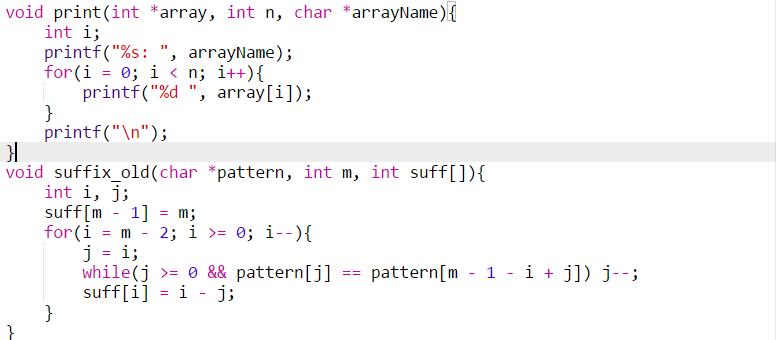
这里是i < m–1不是i < m，也就是最后一个字符如果没有在前面出现过，那么它的bmBc值为m。为什么最后一位不计算在bmBc中呢？很容易想啊，如果记在内**该字符的bmBc就是0，按前所述，pattern需要右移的距离bmBc[‘v’]-m+1+i=-m+1+i <= 0**，也就是**原地不动或走回头路**，当然不干了，前面这种情况已经说的很清楚了，所以这里是m-1。

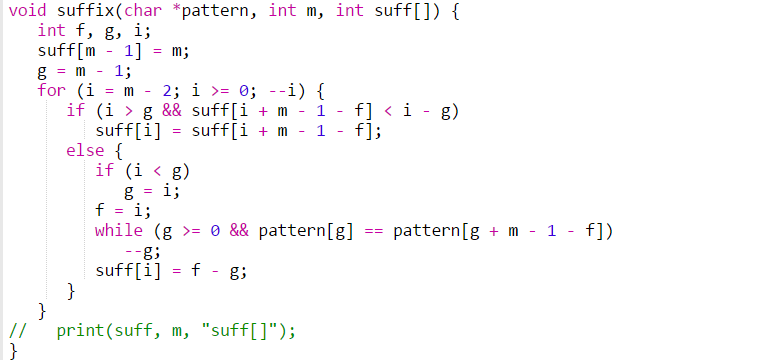
好了，所有的终于都讲完了，下面整合一下这些算法吧

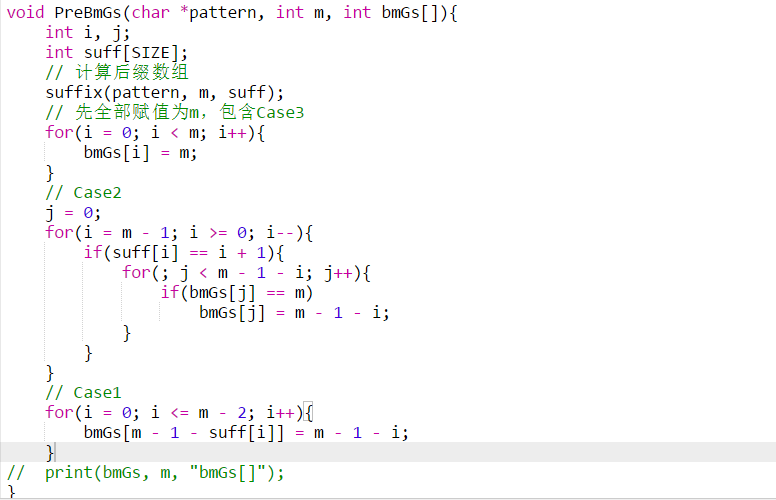
**6、完整代码实现**

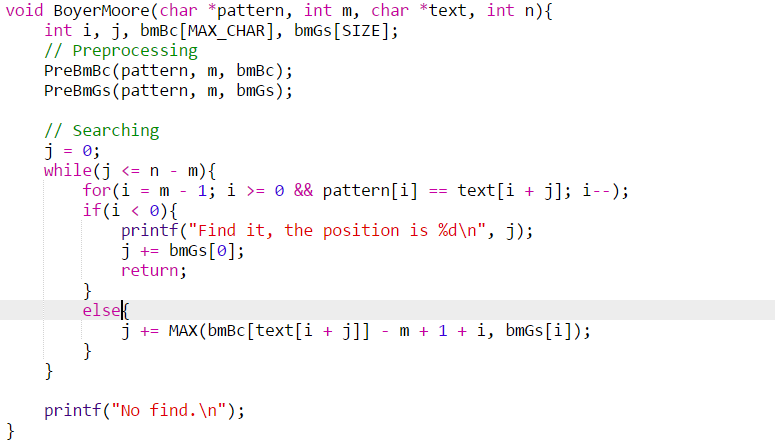












**三、Rabin-Karp算法**

<http://blog.csdn.net/chenhanzhun/article/details/39895077>

<http://www.geeksforgeeks.org/searching-for-patterns-set-3-rabin-karp-algorithm/>

**1、理解介绍**

Rabin-Karp字符串匹配算法和前面介绍的**《朴素字符串匹配算法》**类似，也是***对应每一个字符进行比较***，不同的是Rabin-Karp采用了把字符进行预处理，也就是对每个字符进行对应进制数并取模运算，类似于通过某种函数计算其函数值，比较的是每个字符的函数值。预处理时间O(m)，匹配时间是O((n-m+1)m)。

Rabin-Karp算法的**思想**：

* 假设待匹配字符串的长度为M，目标字符串的长度为N（N>M）；
* 首先计算待匹配字符串的**hash值**，计算目标字符串前M个字符的hash值；
* 比较前面计算的两个hash值，比较次数**N-M+1**：
* 若hash值不相等，则继续计算目标字符串的下一个长度为M的字符子串的hash值
* 若hash值相同，则需要使用**朴素算法再次判断是否为相同的字串**；

该算法在实际运用中，表现不错，**RK算法需要O(m) 时间做预处理**,在最坏情况下，该算法的复杂度与枚举法一样都是，**O((n - m + 1) m)**.但在实际运用中，最坏情况极少出现。

假设**字符集**全是由0到9的数字组成，∑ = {0,1,2…9}。一个长度为k的字符串，例如k=3的字符串”123”, “404”等，都可以看成是***含有k个数字的整形数***，例如前头两个字符串可看做两个整形数：123, 404。由此，对于一个长度为m的字符串P[1…m]，用p表示该字符串对应的含有m个数字的整形数，我们用ts 来表示T[s+1, … , s+m]，这m个字符对应的整形数值，不难发现，***当两个数值相等时，这两个数值对应的字符串就相等***，也就是当且仅当p = ts时，P[1…m] = T[s+1,…,s+m]。**（将字符串匹配问题转换为hash值映射问题）**

把数字从字符串转换为对应的整形数值，我们前头讲过，时间复杂度为O(m).通过下面的公式进行转换即可：

**p = P[m] + 10(P[m-1] + 10(P[m-2]+ …. + ( 10(P[2]) + P[1])…)**

RK算法有一个巧妙之处在于如何*通过ts 去计算ts+1*。假设m=5，T=”314152”，那么可以算出t0 = 31415。t1的数值可以通过一步运算得出： t1 =10\* (t0 - 105−1T[1]) + T[6] = 10(31415 - 105−1 \* 3) + 2 = 14152。

由此可以得到通用公式： **ts+1 =10\* (ts - 10m−1 \* T[s+1]) + T[s + m + 1]，0 <= s <= n - m**

由于一次计算的复杂度是O(1)，计算t0, t1, … , tn−m ,所需要的时间复杂度是O(n - m + 1). 从而，要在 T[1…n] 中查找P[1..m], 所需要的时间复杂度就是O(n - m + 1).

虽说我们当前处理的是**数字字符串**，如果处理的文本是小写字符{a,b…z}, 其实本质是一样的，只要把**十进制的数值**{0,1..9}换成26进制的数字{0, 1, 2, ….25}，那面公式中的10换成26即可。

上面算法，含有一个问题，那就是当m过大，于是对应的数值p或ts过大，会导致溢出，同时就如以前在二进制算法章节中谈到过，***当两个过大的数值比较大小时，CPU需要多个运算周期来进行，这样两数比较***，我们就不能假定他们可以在单位时间内完成了。处理这种情况的处理办法就是**求余**。

**ts+1 = (d\*(ts - T[s+1] \* h ) + T[s+m+1]) mod q**

其中， **h ≡ dm−1 (mod q)**， h的值可以预先通过O(m)次计算得到，q 是一个素数。

然而引入**求余又会引发新的问题**，ts ≡ p (mod q), 并不意味着就一定有 ts = p。但相反，ts ! ≡ p (mod q)，那么一定有 ts ！= p。由此，一旦ts ≡ p (mod q) 满足，我们还需要把T[s+1, … , s+m] 和 P[1…m] 这两个字符串**逐个字符比较**，每个字符都一样，才能最终断定T[s+1,…,s+m] = P[1…m].

这就解释了为何RK算法最坏情况下，复杂度会是**O(m (n - m + 1))**. 因为有可能出现这样情况，ts ≡ p (mod q)，但T[s+1, … s+m] 与 P[1…m]不匹配。

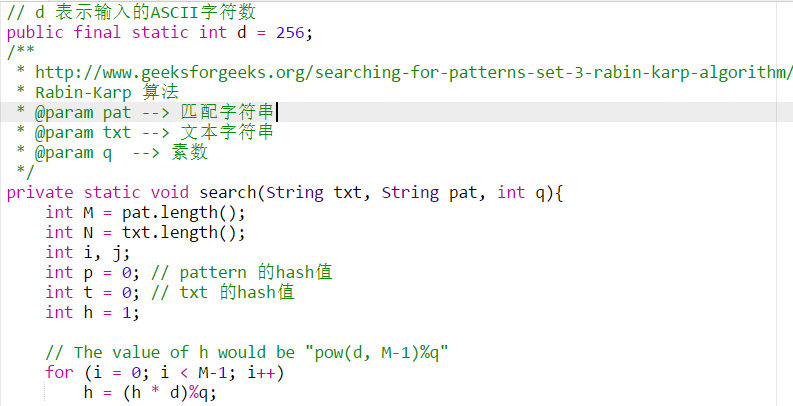
**举个具体例子看看：**

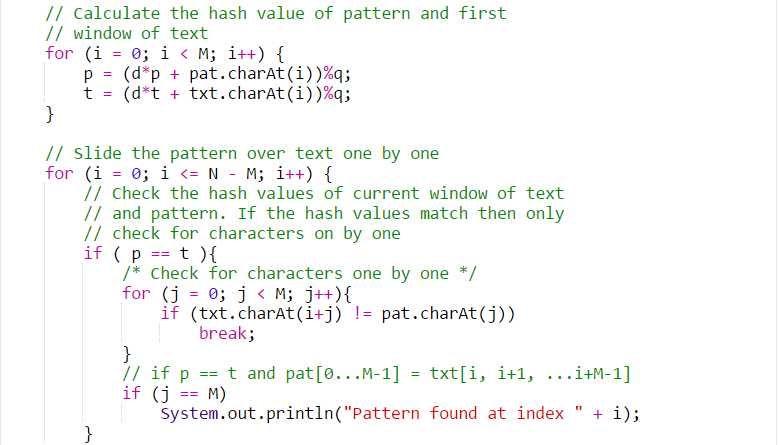
T = “2359023141526739921，q = 13，d = 10，P=31415，m=5，不难发现s = 6 的时候，满足T[s+1, … s+m+1] = P[1..5], 但是当s=12时，T[s+1, …, s+m+1] = “67399”, 然而7 ≡ 67399 ≡ 31415 (mod 13). 但是字符串”67399” 与字符串 “31415”并不匹配。

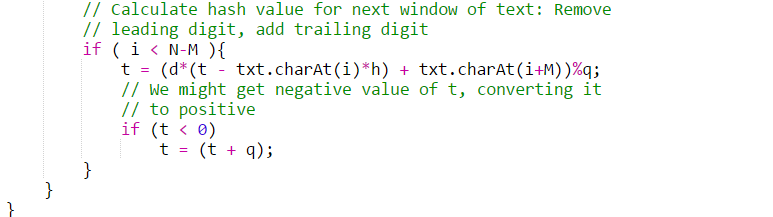
**2、伪代码**

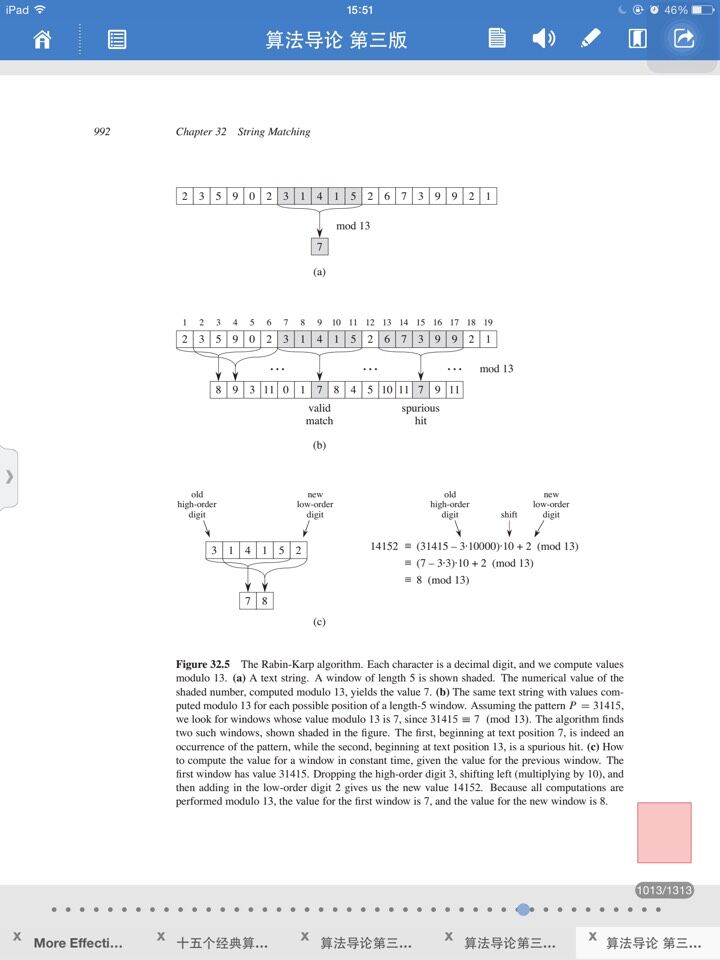


**3、源代码**









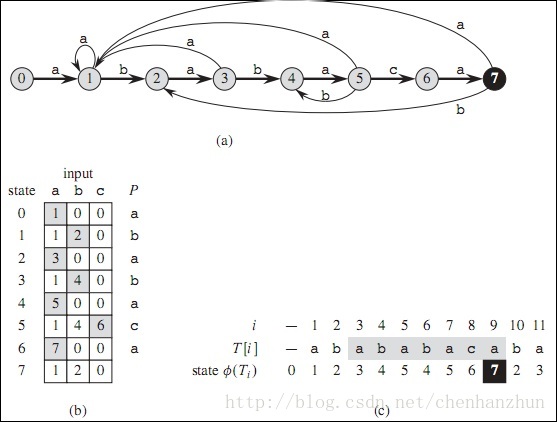
**四、有限自动机的字符串匹配算法**

<http://blog.csdn.net/weishenmetlc/article/details/51511483>

**1、理解与介绍**

有限自动机(Finite Automata)字符串匹配算法最主要的是**计算出转移函数**。即给定一个当前状态k和一个字符x，计算下一个状态；计算方法为：**找出模式pat的最长前缀prefix**，同时也是pat[0...k-1] x(注意：字符串下标是从0开始)的后缀，则prefix的长度即为下一个状态。匹配的过程是比较输入文本子串和模式串的状态值，若相等则存在，若不相等则不存在。

模式串为**P=abababaca**的自动机执行过程：**（有限自动机过程）**



**例子：对模式 P = aabab构造出相应的字符串匹配自动机，并说明它在文本字符串T=aaababaabaababaab上的操作过程。**

再讲这个例子之前，我们有必要先来了解一下自动机是什么意思？

有限自动机是什么意思？是一个**处理信息的简单机器**，通过*对文本字符串T进行扫描，找出模式P的所有出现的位置*。它们**只对每个文本字符检查一次，并且检查每个文本字符时所用的时间为常数**。

我们来看看它的伪代码：



可以看出，他的时间复杂度为o(n),但是，这个匹配时间，并没有包括计算转移函数ξ的预处理时间。接下来，我们来做一做上面那个题目。

**有限自动机**分为5个组员（Q、q=0、A、∑、ξ）

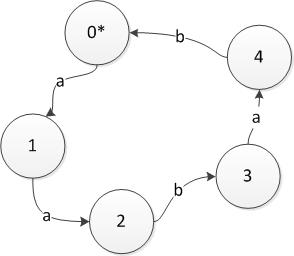
我们假设：Q = {0,1,2,3,4,99} //99用来表示不会取得，如果等于99就跳出循环。

Q：初始化为0

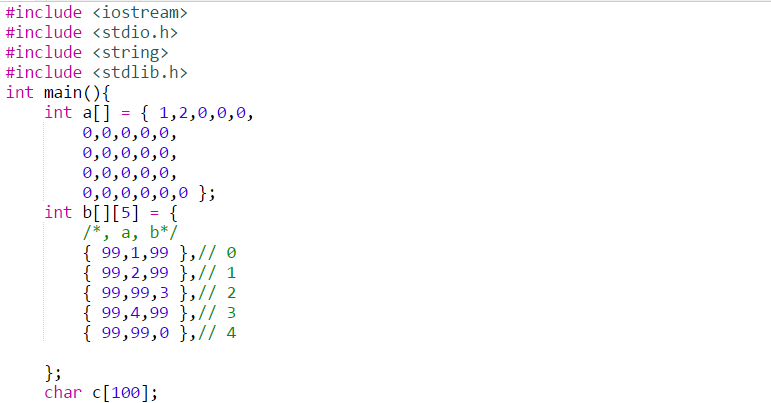
A={0} //代表了终点，这里只有一个终点

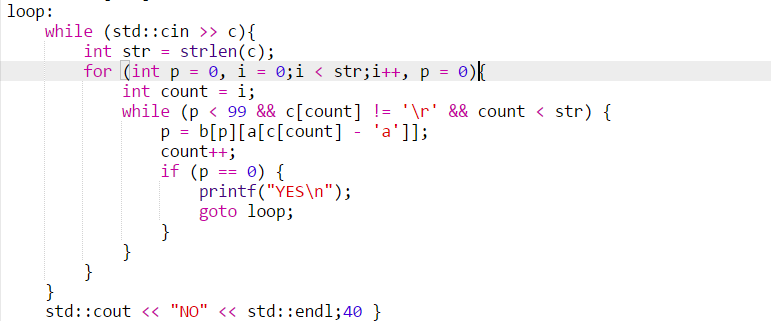
∑={a,b,c,d....x,y,z}

关于函数，我设计成这样：

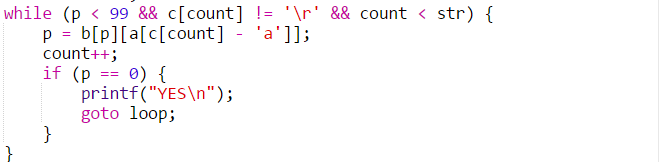


**aabab：{0\* -> 1 -> 2 -> 3 -> 4 -> 0\*}**





我们先创建一个数组a，因为**字符串aabab**只有a和b，所以初始化依次从1递增，其他为0。二维数据b，保存p值。



这段代码，用于**获取p值。也就是ξ函数**。也许，我们需要把他单独摘出来，要不然这个函数的执行时间为O（n\*m）（展开下列函数，可看到和伪代码相同的c++代码）,但事实上，他已经比BF算法好太多了。当然他也有弊病，如果预处理的时间太长，该怎么办？这是一个值得考虑的问题。换句话说，如果∑特别多，我们这里只有2个，建立的自动机时间也很长。我们有方法处理。

**2、介绍（再次）**

<https://my.oschina.net/amince/blog/182210>

**1）有限自动机定义及基本术语：**

一个有限自动机 M 是一个**5元组**（Q、q0、A、Σ、δ），其中：

* Q 是所有状态的有限集合;
* q0 ∈ Q (属于)是**初始状态**;
* A ⊆ Q （子集）是接**受状态的集合**;(对应于多模式？)
* Σ 是有限输入字母表;
* δ 是从Q \* Σ的**转移函数**，称为有限自动机M的转移函数;

**记号与术语：**

* Σ\*：表示用字母表Σ中所有字符形成的*所有有限长度的字符串集合*.
* n：输入字符串(input string)的长度.
* m：模式字符串(pattern string)的长度；也称作终态m，当状态为m时表示，m长度的模式串匹配成功.
* |x| ：字符串x的长度, 如示符号记法.
* ：字符串w 是字符串x 的**前缀**，如示符号记法.
* ：字符串w 是字符串x 的**后缀**，如示符号记法.（注意前缀/后缀均遵循传递规则）
* ε：表示**空字符串**，是所有字符串的后缀、前缀. (ε读作 epsilon )
* a ：下文中的字符a泛指所有字符(a∈Σ)，不特指字符'a'.

**2）引入的函数定义**

* **转移函数δ（transition function）** ( δ 读作"delta",对应大写为 Δ )。有限自动机开始于初始状态q0，***每次读入输入字符串的一个字符***，如果有限自动机在状态q是读入字符'a'，则M状态从q变成 δ(q, a)；
* **终态函数 Φ（finite state function）**( Φ 读作"fai", 对应小写为 φ )。是从 Σ\* 到 Q 的函数，Φ(w)是自动机M 扫描字符串 w 终止后的状态；M 接受字符串w，当且仅当Φ(w)∈A, 函数Φ有下列递归关系定义：

φ(ε) = q0;（空字符串 ε 的终态为q0）

φ(wa) = δ(φ(w),a) (其中w∈Σ\*，a∈Σ)

* **辅助函数，后缀函数σ**对应于模式字串P ( σ 读作 "sigma", 对应大写为 Σ )。是从Σ\* 到{0,1, ..., m}上的映射，***σ(x)是字符串x的后缀同时是P的前缀的最大长度***；

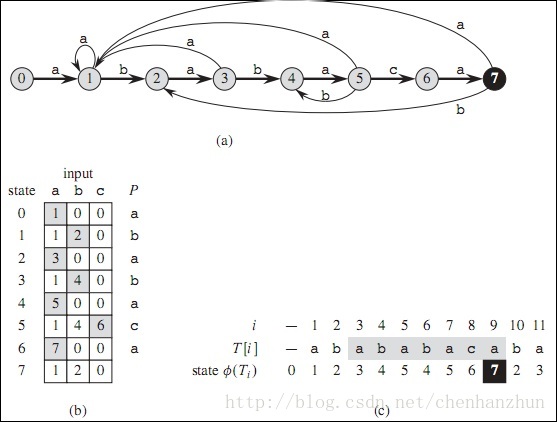
σ(x) = max{k: Pk ⊐ x }

有P0 = ε是所有所有字符串的后缀;

\* 后缀函数的主要意义的是求出当前匹配失败时，求出已经匹配过的**部分字串x**是否是待匹配模式字串P的**前缀**，即匹配可以跳过x中部分长度( σ(x) ),可以用于实现转移过程;同时也表明**在接受输入字符串x后的状态(终态)**，即也用于实现终态函数。

**3）字符串匹配自动机(string-matching automation)**

下图是依据模式串 P="ababaca" 构建的自动机图表：



上图(a)是一个自动机的状态转换图表,接受所有以字符串"ababaca"结尾的字符串。其中状态0是初始状态，状态7是**唯一接受状态(**单模式匹配).

* 1)从状态i到状态j的带箭头的**有向边**表示转移过程: δ(i, a) = j(a∈Σ).
* 2) **右向边组成了自动机的主要"骨架"**，图中粗线部分，对应于*输入字符同模式字串匹配成功的转移过程*。左向边对应于*匹配失败的转移过程*(**跳转**，主要是计算已经匹配的部分字串的后缀子串同时是模式串P的前缀的**最大长度**)。部分匹配失败的过程没有标示出来。
* 3) 图中部分状态i在接受某字符a(a∈Σ)时，**没有标示出对应有向边的情况表明其转移过程为: δ(i, a) = 0(a∈Σ)**，根据下面字符串模式匹配自动机定义，可知当前已经匹配子串没有后缀字串是模式串P的前缀。如在状态3时，输入字符为'c'，即在已经匹配了"aba"这时接受字符'c'，可知当前已匹配字串为"abac"，对应模式字串P="ababaca"，可知这时**匹配失败**，进行失败***跳转求"abac"后缀子串同时是模式串P前缀的最大长度***，可知为0.
* 4) 匹配成功的转移过程(对应状态，以及对应输入字符)均标示为灰色,
* 5) 表(c)是自动机在处理(接受)输入文本T="abababacaba"的**最终状态表**。当输入字符T[i]时，此时字串T[0...i]对应的的**最终状态φ(T[0...i])** 同表(c)最后一列一一对应。有T["abababaca"] = P.length = 7(唯一接受状态)，即这时候在T串中匹配成功模式串P，结束位置为9，起始位置为(9-P.length+1)=3。

**字符串匹配有限自动机定义：**

给定模式(pattern)字符串 P[1...m]，其对应的字符串匹配有限自动机定义如下:

1、状态集Q = {0,1,...m}，开始状态q0 是状态0，state m 是**唯一的接受状态**；

2、转移函数δ 可以用**后缀函数来表示** (这个很重要，因为状态转移函数是个抽象概念，而后缀函数可以用code表示) ：

**δ(q,a) = σ（Pq,a） <等式一>**

假设当前已经**读入的字符串为T**，为了让T的字串(以T[i]为结尾) 能匹配模式字串Pj，必须满足**Pj是Ti的后缀**；同时假设q = φ(Ti)，说明读取字串Ti后自动机M 状态变成q；同时根据转移函数<等式一>，可知***q是模式字串P最大长度的前缀，同时是Ti的后缀***；因此在状态q，有Pq ⊐ Ti 和 q = σ(Ti) （当q 等于m 时，说明模式字串P整个是Ti的后缀，也意味着匹配查找成功了），因此有σ(Ti)= q，得出永动机也支持下面的等式(终态函数也是抽象的，转化为后缀函数表达式后，可以用code表示)：

**φ(Ti) = σ(Ti)（i = 0,1,...n） <等式二>**

同时有两个引理(具体证明可以参考算法导论)：

* 引理1、后缀函数不等式：

σ(xa) ≤ σ(x) + 1 (对于任何字符串x，以及字母a)

* 引理2、后缀函数递归引理：

对于任何字符串x，以及字母a，如果q = σ(x)，有：**σ(xa) = σ(Pqa)**

<等式二> 可以用数学归纳法证明，具体如下:

1、当i = 0，因为T0 = ε，因此有φ(Ti) = 0 = σ(Ti)

2、假设φ(Ti) = σ(Ti)，证明φ(Ti+1) = σ(Ti+1)，用q 表示φ(Ti)，用字母a表示T[i+1],有：

φ(Ti+1) = φ(Tia) (Ti+1 == Tia) = δ(φ(Ti),a) (根据终态函数的定义)

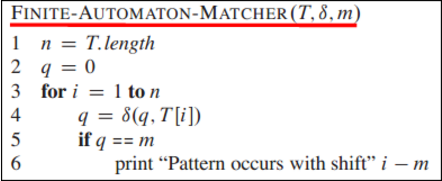
= δ(q,a) （根据q的定义）

= σ(Pqa) （根据等式一）

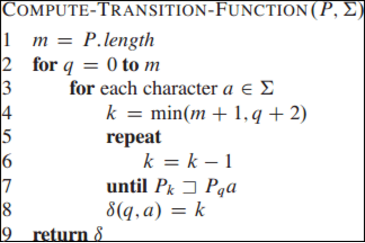
= σ(Tia) (根据引理二)

= σ(Ti+1) (Ti+1 == Tia)

从上面可以知道当**读入Ti的终态**(亦即读入T[i]后***转移函数状态***)等于**模式长度**，就匹配成功了，下面是有限自动机机匹配算法伪代码：



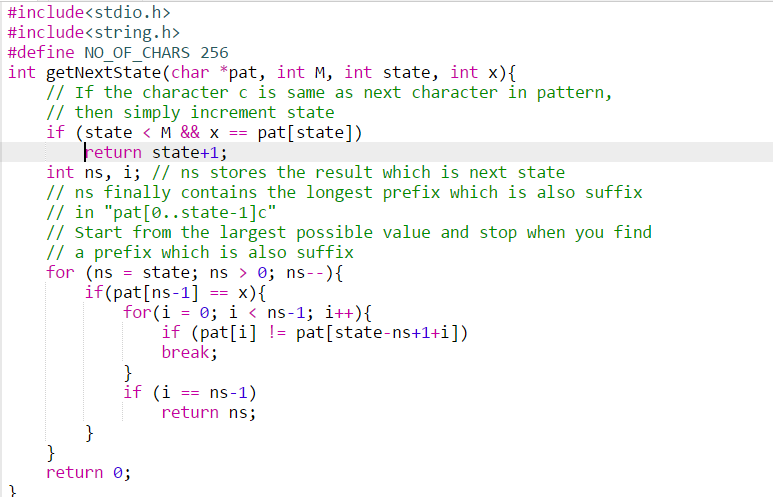
下面就是根据**<等式一>**来**实现转移函数**的伪代码：

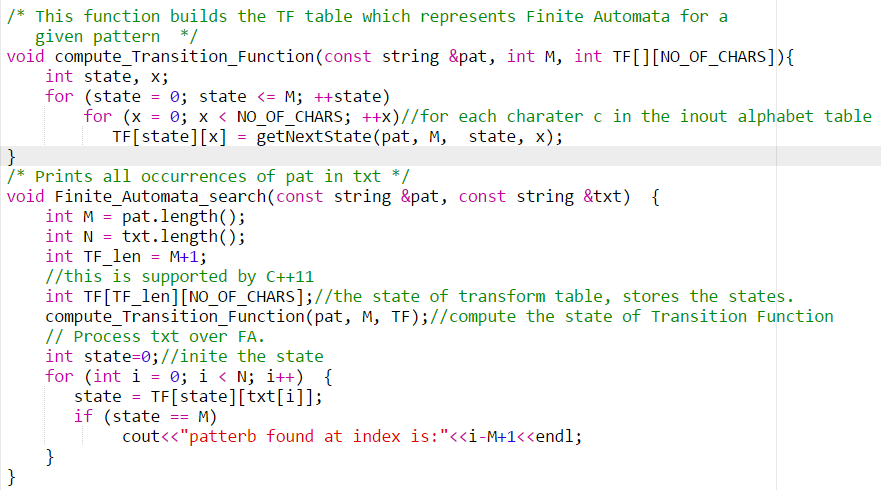


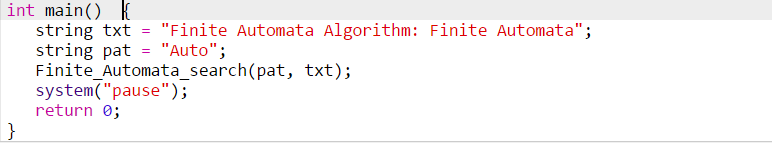
**4）完整代码**

<http://www.geeksforgeeks.org/searching-for-patterns-set-5-finite-automata/>

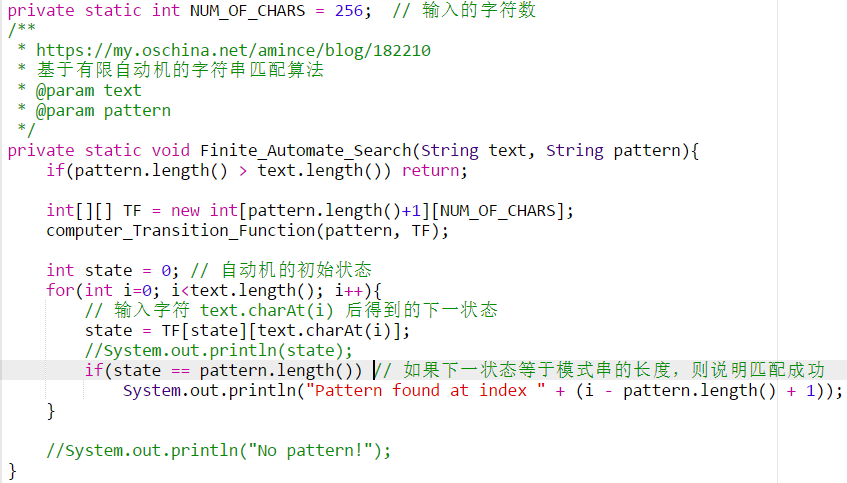
**A. C++代码**







**B. Java代码**





**五、Sunday 模式匹配算法**

**1、原理1**

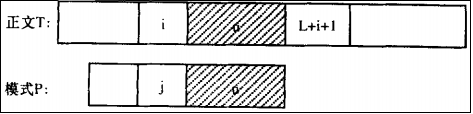
<https://my.oschina.net/amince/blog/182570>

**1）概念介绍**

Sunday算法是Daniel M.Sunday于1990年提出的一种比BM算法搜索速度更快的算法。该算法的**核心思想**是：在*匹配过程中，模式串并不被要求一定要按从左向右进行比较还是从右向左进行比较，它在发现不匹配时，算法能跳过尽可能多的字符以进行下一步的匹配*，从而提高了匹配效率。

相对于**KMP算法与BM算法原理**复杂不易实现，Sunday算法原理简单执行速度快。本文介绍下该算法的实现，以及该算法一些的改进

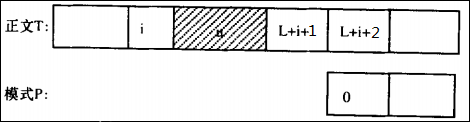
假设在发生不匹配时，**T[i] ≠ P[j](0≤i<N, 0≤j<m)**， 此时已经匹配的部分字符串为u,并假设**子串u的长度为L**。如下图。



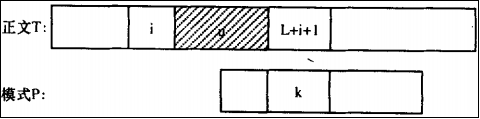
这时，我们可以知道**T[L+i+l]肯定要参加下一轮的匹配**，因为P[m-1]至少要移动到这个位置(即模式串P至少向右移动一个字符的位置)。

下面有两种情况：

* T[L+i+1]在模式串P中没有出现：这个时候模式申**P移动到T[L+i+1]**(包括T[L+i+1]本身)之前的任何位置，都没有意义。因此，P[0]必须移动到T[L+i+1]之后的**字符的位置**；如图：



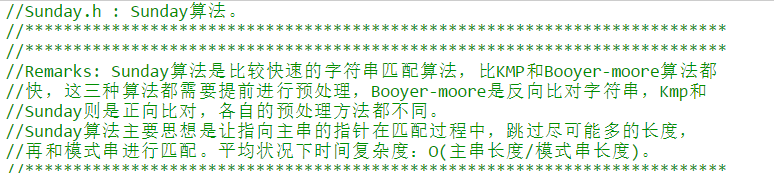
* T[L+i+1]有在模式串P中出现：指T[L+i+1]从**模式串P的右侧**，按P[m-1]、P [m-2]...P[0]的次序查找(字符在模式串中最右边出现位置)。如果发现T[L+i+1]和P中的某个字符相同，则记下这个位置，记为k(0 < k < m)；如图：

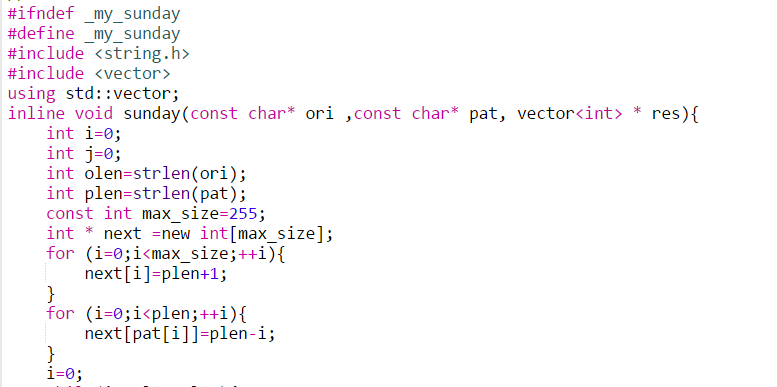


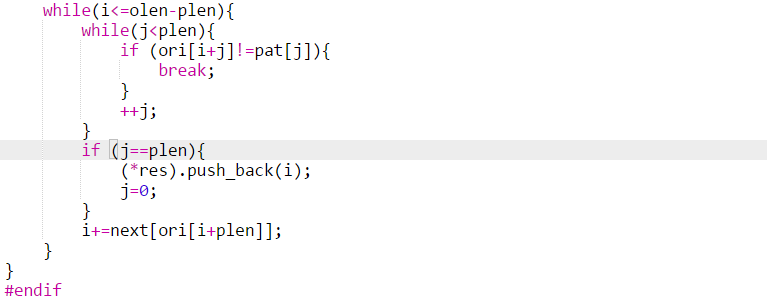
按照这种移动算法，然后***按照字符从左向右的次序匹配***。如果完全匹配了，则匹配成功；否则直接跳过，一直匹配到**主串最右端**；

**2）代码**

<http://www.oschina.net/code/snippet_170948_12386>







**2、原理2**

<http://www.cnblogs.com/lbsong/archive/2012/05/25/2518188.html>

**1）介绍**

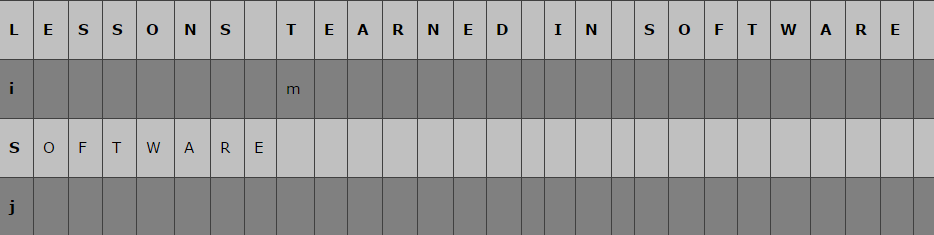
假设我们有如下字符串：

A = "LESSONS TEARNED IN SOFTWARE TE";

B = "SOFTWARE";

**Sunday算法的大致原理：**

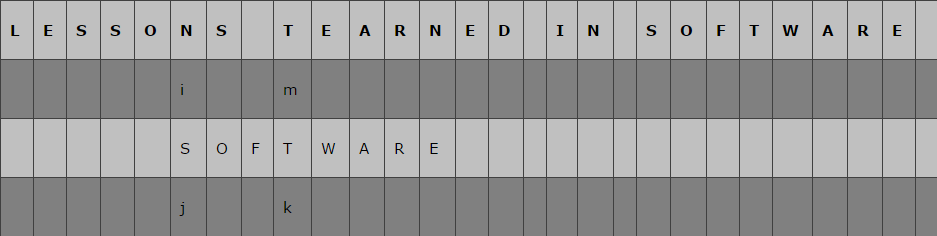
先**从左到右逐个字符比较**，以我们的字符串为例：开始的时候，我们让i = 0, 指向A的第一个字符; j = 0 指向B的第一个字符，分别为"L"和"S"，不等；这个时候，Sunday算法要求，找到位于A字串中位于B字符串后面的第一个字符，即下图中 m所指向的字符"T"，在模式字符串B中**从后向前**查找是否存在"T"，



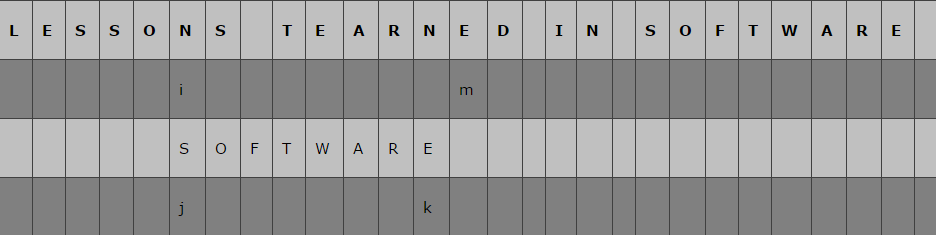
可以看到下图中k指向的字符与m指向的字符相等，



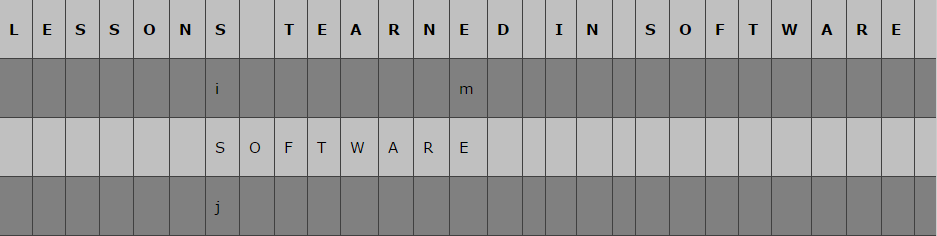
这时就将相等的字符对齐，让j再次指向B字符串的头一个字符，相应地，将*i指向主串对应的字符N*



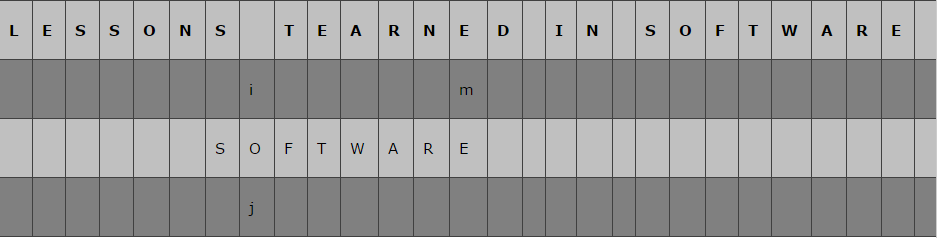
再次比较A[i]和B[j]，不等，这时*再次寻找主串中在模式串后面的那个字符*



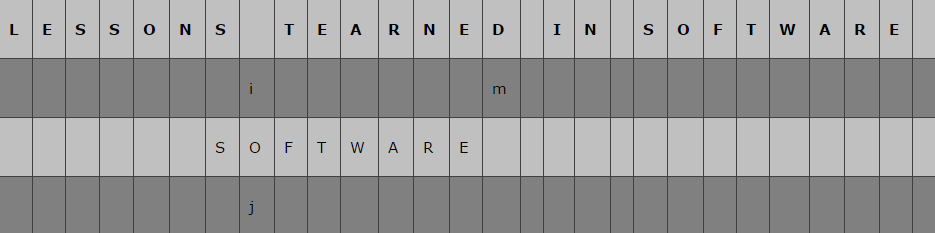
我们看到，模式串的最后一个字符与m指向的主串字符相等，因此再次移动子串



这时，主串i对应的字符是S，j对应的子串字符也是S，i++, j++



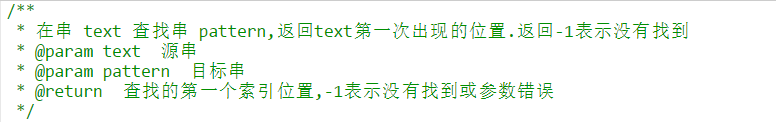
现在再次不等，m指向字符"D"

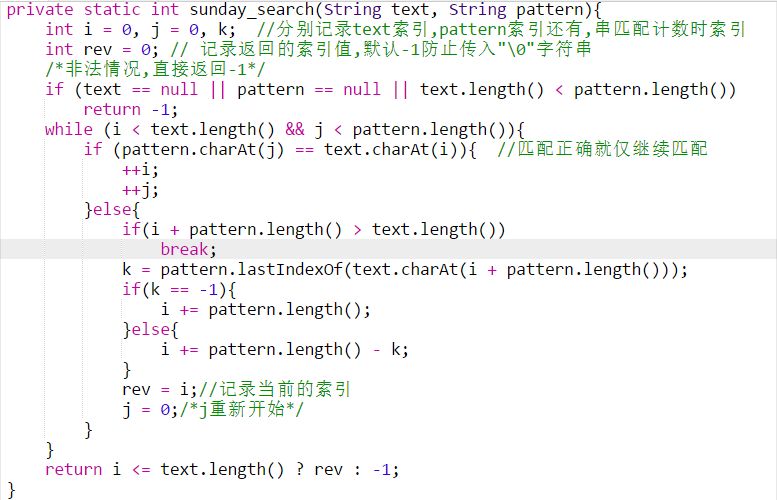


….

直到找到，或者i到达主串的末尾

**2）代码**





<http://www.jianshu.com/p/2594d312cefd>

