从头到尾彻底理解KMP

极客学院出版

前言

本文出自 七月在线科技创始人兼 CEO，结构之法算法之道 blog 之博主。

版权信息

> 作者：July
>
> 时间：最初写于 2011 年 12 月，2014 年 7 月 21 日晚 10 点 全部删除重写成此文，随后的半个多月不断反复改进。
>
> 原文链接：http://blog.csdn.net/v_july_v/article/details/7041827

目录

1. 引言

本 KMP 原文最初写于 2 年多前的 2011 年 12 月，因当时初次接触 KMP，思路混乱导致写也写得混乱。所以一直想找机会重新写下 KMP，但苦于一直以来对 KMP 的理解始终不够，故才迟迟没有修改本文。

然近期因在北京开了个算法班，专门讲解数据结构、面试、算法，才再次仔细回顾了这个 KMP，在综合了一些网友的理解、以及跟我一起讲算法的两位讲师朋友曹博、邹博的理解之后，写了 9 张 PPT，发在微博上。随后，一不做二不休，索性将 PPT 上的内容整理到了本文之中（后来文章越写越完整，所含内容早已不再是九张 PPT 那样简单了）。

KMP 本身不复杂，但网上绝大部分的文章（包括本文的 2011 年版本）把它讲混乱了。下面，咱们从暴力匹配算法讲起，随后阐述 KMP 的流程 步骤、next 数组的简单求解 递推原理 代码求解，接着基于 next 数组匹配，谈到有限状态自动机，next 数组的优化，KMP 的时间复杂度分析，最后简要介绍两个 KMP 的扩展算法。

全文力图给你一个最为完整最为清晰的 KMP，希望更多的人不再被 KMP 折磨或纠缠，不再被一些混乱的文章所混乱，有何疑问，欢迎随时留言评论，thanks。

2

2. 暴力匹配算法

假设现在我们面临这样一个问题：有一个文本串 S，和一个模式串 P，现在要查找 P 在 S 中的位置，怎么查找呢？

如果用暴力匹配的思路，并假设现在文本串 S 匹配到 i 位置，模式串 P 匹配到 j 位置，则有：

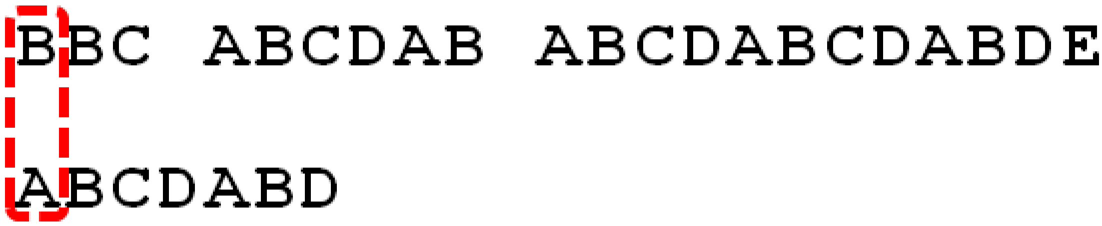
- 如果当前字符匹配成功（即 S[i] == P[j]），则 i++，j++，继续匹配下一个字符；
- 如果失配（即 S[i]! = P[j]），令 i = i - (j - 1)，j = 0。相当于每次匹配失败时，i 回溯，j 被置为 0。

理清楚了暴力匹配算法的流程及内在的逻辑，咱们可以写出暴力匹配的代码，如下：

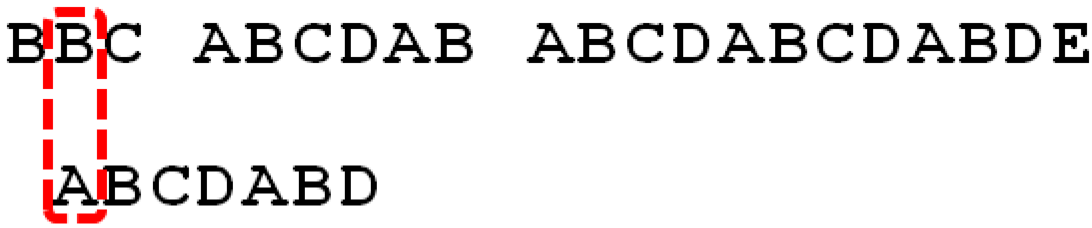
```
int ViolentMatch(char* s, char* p)
{
    int sLen = strlen(s);
    int pLen = strlen(p);

    int i = 0;
    int j = 0;
    while (i < sLen && j < pLen)
    {
        if (s[i] == p[j])
        {
            //①如果当前字符匹配成功（即S[i] == P[j]），则i++，j++
            i++;
            j++;
        }
        else
        {
            //②如果失配（即S[i]! = P[j]），令i = i - (j - 1)，j = 0
            i = i - j + 1;
            j = 0;
        }
    }
    //匹配成功，返回模式串p在文本串s中的位置，否则返回-1
    if (j == pLen)
        return i - j;
    else
        return -1;
}
```

举个例子，如果给定文本串 S“BBC ABCDAB ABCDABCDABDE”，和模式串 P“ABCDABD”，现在要拿模式串 P 去跟文本串 S 匹配，整个过程如下所示：

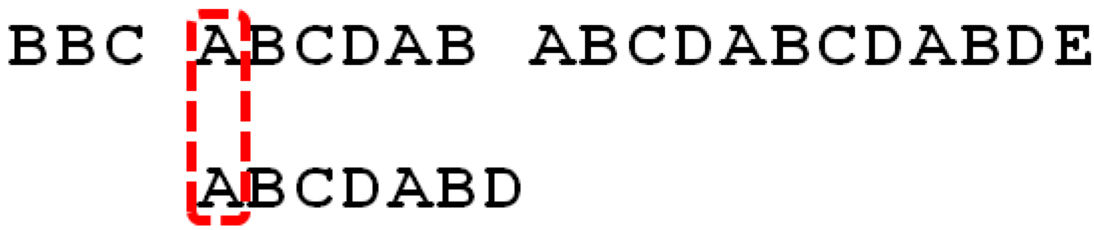
1.S[0] 为 B，P[0] 为 A，不匹配，执行第 ② 条指令：“如果失配（即 S[i]! = P[j]），令 i = i - (j - 1)，j = 0”，S[1] 跟 P[0] 匹配，相当于模式串要往右移动一位（i=1，j=0）



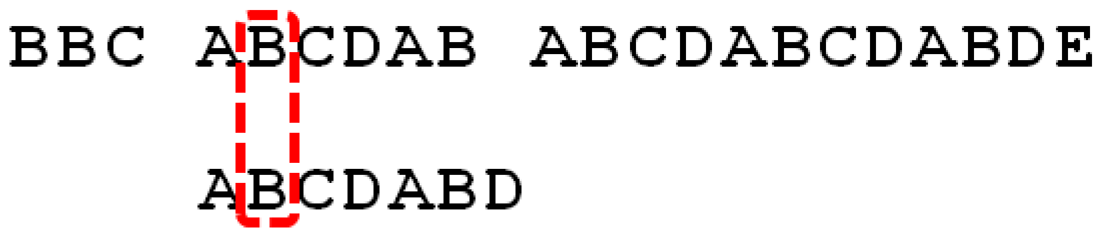
2.S[1] 跟 P[0] 还是不匹配，继续执行第 ② 条指令：“如果失配（即 S[i]! = P[j]），令 i = i - (j - 1)，j = 0”，S[2] 跟 P[0] 匹配（i=2，j=0），从而模式串不断的向右移动一位（不断的执行“令 i = i - (j - 1)，j = 0”，i 从 2 变到 4，j 一直为 0）



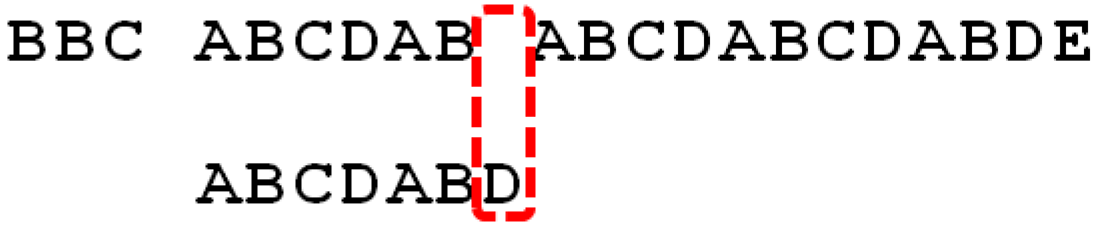
3.直到 S[4] 跟 P[0] 匹配成功（i=4，j=0），此时按照上面的暴力匹配算法的思路，转而执行第 ① 条指令：“如果当前字符匹配成功（即 S[i] == P[j]），则 i++，j++”，可得 S[i] 为 S[5]，P[j] 为 P[1]，即接下来 S[5] 跟 P[1] 匹配（i=5，j=1）



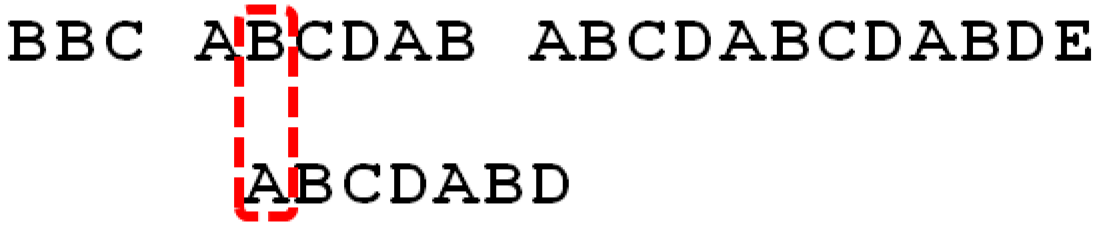
4.S[5] 跟 P[1] 匹配成功，继续执行第 ① 条指令：“如果当前字符匹配成功（即 S[i] == P[j]），则 i++，j++”，得到 S[6] 跟 P[2] 匹配（i=6，j=2），如此进行下去



5.直到 S[10] 为空格字符，P[6] 为字符 D（i=10，j=6），因为不匹配，重新执行第 ② 条指令：“如果失配（即 S[i]! = P[j]），令 i = i - (j - 1)，j = 0”，相当于 S[5] 跟 P[0] 匹配（i=5，j=0）



6.至此，我们可以看到，如果按照暴力匹配算法的思路，尽管之前文本串和模式串已经分别匹配到了 S[9]、P[5]，但因为 S[10] 跟 P[6] 不匹配，所以文本串回溯到 S[5]，模式串回溯到 P[0]，从而让 S[5] 跟 P[0] 匹配。



而 S[5] 肯定跟 P[0] 失配。为什么呢？因为在之前第 4 步匹配中，我们已经得知 S[5] = P[1] = B，而 P[0] = A，即 P[1] != P[0]，故 S[5] 必定不等于 P[0]，所以回溯过去必然会导致失配。那有没有一种算法，让 i 不往回退，只需要移动 j 即可呢？

答案是肯定的。这种算法就是本文的主旨 KMP 算法，它利用之前已经部分匹配这个有效信息，保持 i 不回溯，通过修改 j 的位置，让模式串尽量地移动到有效的位置。

3

3. KMP 算法

3.1 定义

Knuth-Morris-Pratt 字符串查找算法，简称为 “KMP 算法”，常用于在一个文本串 S 内查找一个模式串 P 的出现位置，这个算法由 Donald Knuth、Vaughan Pratt、James H. Morris 三人于 1977 年联合发表，故取这三人的姓氏命名此算法。

下面先直接给出 KMP 的算法流程（如果感到一点点不适，没关系，坚持下，稍后会有具体步骤及解释，越往后看越会柳暗花明 ?）：

- 假设现在文本串 S 匹配到 i 位置，模式串 P 匹配到 j 位置
  - 如果 j = -1，或者当前字符匹配成功（即 S[i] == P[j]），都令 i++，j++，继续匹配下一个字符；
  - 如果 j != -1，且当前字符匹配失败（即 S[i] != P[j]），则令 i 不变，j = next[j]。此举意味着失配时，模式串 P 相对于文本串 S 向右移动了 j - next [j] 位。
    - 换言之，当匹配失败时，模式串向右移动的位数为：失配字符所在位置 - 失配字符对应的 next 值（next 数组的求解会在下文的 3.3.3 节 (页 16)中详细阐述），**即移动的实际位数为：j - next[j]**，且此值大于等于1。

很快，你也会意识到 next 数组各值的含义：代表当前字符之前的字符串中，有多大长度的相同前缀后缀。例如，如果 next [j] = k，代表 j 之前的字符串中有最大长度为 k 的相同前缀后缀。

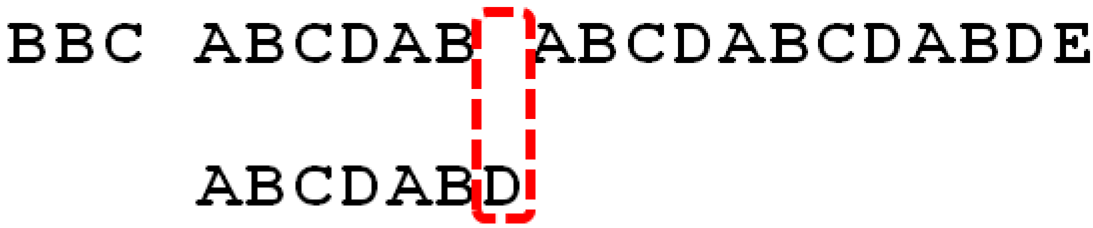
此也意味着在某个字符失配时，该字符对应的 next 值会告诉你下一步匹配中，模式串应该跳到哪个位置（跳到next [j] 的位置）。如果 next [j] 等于 0 或 -1，则跳到模式串的开头字符，若 next [j] = k 且 k > 0，代表下次匹配跳到 j 之前的某个字符，而不是跳到开头，且具体跳过了 k 个字符。

转换成代码表示，则是：

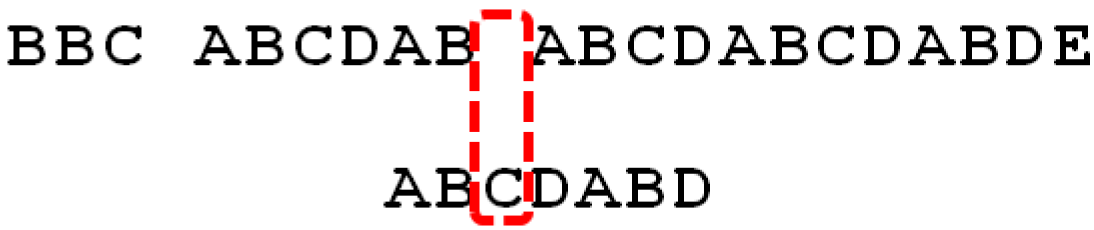
```
int KmpSearch(char* s, char* p)
{
    int i = 0;
    int j = 0;
    int sLen = strlen(s);
    int pLen = strlen(p);
    while (i < sLen && j < pLen)
    {
        //①如果j = -1，或者当前字符匹配成功（即S[i] == P[j]），都令i++，j++
        if (j == -1 || s[i] == p[j])
        {
            i++;
```

```
            j++;
        }
        else
        {
            //②如果j != -1，且当前字符匹配失败（即S[i] != P[j]），则令 i 不变，j = next[j]
            //next[j]即为j所对应的next值
            j = next[j];
        }
    }
    if (j == pLen)
        return i - j;
    else
        return -1;
}
```

继续拿之前的例子来说，当 S[10] 跟 P[6] 匹配失败时，KMP 不是跟暴力匹配那样简单的把模式串右移一位，而是执行第 ② 条指令：“如果 j != -1，且当前字符匹配失败（即 S[i] != P[j]），则令 i 不变，j = next[j]”，即 j 从 6 变到 2（后面我们将求得 P[6]，即字符 D 对应的 next 值为 2），所以相当于模式串向右移动的位数为 j - next[j]（j - next[j] = 6-2 = 4）。



向右移动 4 位后，S[10] 跟 P[2] 继续匹配。为什么要向右移动 4 位呢，因为移动 4 位后，模式串中又有个“AB”可以继续跟 S[8]S[9] 对应着，从而不用让 i 回溯。相当于在除去字符 D 的模式串子串中寻找相同的前缀和后缀，然后根据前缀后缀求出next 数组，最后基于 next 数组进行匹配（不关心 next 数组是怎么求来的，只想看匹配过程是咋样的，可直接跳到下文 3.3.4 节 (页 18)）。



3.2 步骤

- ① 寻找前缀后缀最长公共元素长度

对于 P = p0 p1 ...pj-1 pj，寻找模式串 P 中长度最大且相等的前缀和后缀。如果存在 p0 p1 ...pk-1 pk = pj- k pj-k+1...pj-1 pj，那么在包含 pj 的模式串中有最大长度为 k+1 的相同前缀后缀。举个例子，如果给定的模式串为“abab”，那么它的各个子串的前缀后缀的公共元素的最大长度如下表格所示：

模式串	a	b	a	b
最大前缀后缀公共元素长度	0	0	1	2

比如对于字符串 aba 来说，它有长度为 1 的相同前缀后缀 a；而对于字符串 abab 来说，它有长度为 2 的相同前缀后缀ab（相同前缀后缀的长度为 k + 1，k + 1 = 2）。

- ② 求 next 数组

next 数组考虑的是除当前字符外的最长相同前缀后缀，所以通过第 ① 步骤求得各个前缀后缀的公共元素的最大长度后，只要稍作变形即可：将第 ① 步骤中求得的值整体右移一位，然后初值赋为 -1，如下表格所示：

模式串	a	b	a	b
next数组	-1	0	0	1

比如对于 aba 来说，第 3 个字符 a 之前的字符串 ab 中有长度为 0 的相同前缀后缀，所以第 3 个字符 a 对应的 next 值为 0；而对于 abab 来说，第 4 个字符 b 之前的字符串 aba 中有长度为 1 的相同前缀后缀 a，所以第 4 个字符 b 对应的 next 值为 1（相同前缀后缀的长度为 k，k = 1）。

- ③ 根据 next 数组进行匹配

匹配失配，j = next [j]，模式串向右移动的位数为：j - next[j]。换言之，当模式串的后缀 pj-k pj-k+1, ..., pj-1 跟文本串 si-k si-k+1, ..., si-1 匹配成功，但 pj 跟 si 匹配失败时，因为 next[j] = k，相当于在不包含 pj 的模式串中有最大长度为 k 的相同前缀后缀，即 p0 p1 ...pk-1 = pj-k pj-k+1...pj-1，故令 j = next[j]，从而让模式串右移 j - next[j] 位，使得模式串的前缀 p0 p1, ..., pk-1 对应着文本串 si-k si-k+1, ..., si-1，而后让 pk 跟 si 继续匹配。如下图所示：

综上，KMP 的 next 数组相当于告诉我们：当模式串中的某个字符跟文本串中的某个字符匹配失配时，模式串下一步应该跳到哪个位置。如模式串中在j 处的字符跟文本串在 i 处的字符匹配失配时，下一步用 next [j] 处的字符继续跟文本串 i 处的字符匹配，相当于模式串向右移动 j - next[j] 位。

接下来，分别具体解释上述 3 个步骤。

3.3 解释

3.3.1 寻找最长前缀后缀

如果给定的模式串是：“ABCDABD”，从左至右遍历整个模式串，其各个子串的前缀后缀分别如下表格所示：

模式串的各个子串	前缀	后缀	最大公共元素长度
A	空	空	0
AB	A	B	0
ABC	A,AB	C,BC	0
ABCD	A,AB,ABC	D,CD,BCD	0
ABCDA	A,AB,ABC,ABCD	A,DA,CDA,BCDA	1
ABCDAB	A,AB,ABC,ABCD,ABCDA	B,AB,DAB,CDAB,BCDAB	2
ABCDABD	A,AB,ABC,ABCD,ABCDA ABCDAB	D,BD,ABD,DABD,CDABD BCDABD	0

也就是说，原模式串子串对应的各个前缀后缀的公共元素的最大长度表为（下简称《最大长度表》）：

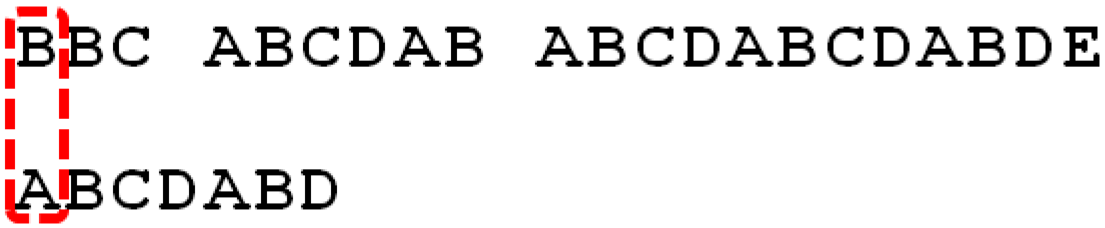
字符	A	B	C	D	A	B	D
最大前缀后缀公共元素长度	0	0	0	0	1	2	0

3.3.2 基于《最大长度表》匹配

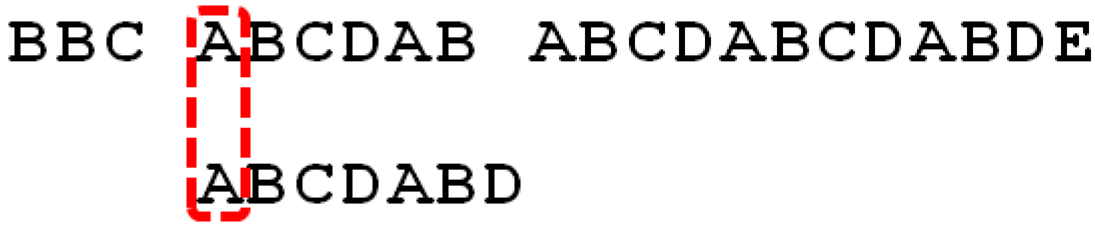
因为模式串中首尾可能会有重复的字符，故可得出下述结论：

失配时，模式串向右移动的位数为：已匹配字符数 - 失配字符的上一位字符所对应的最大长度值

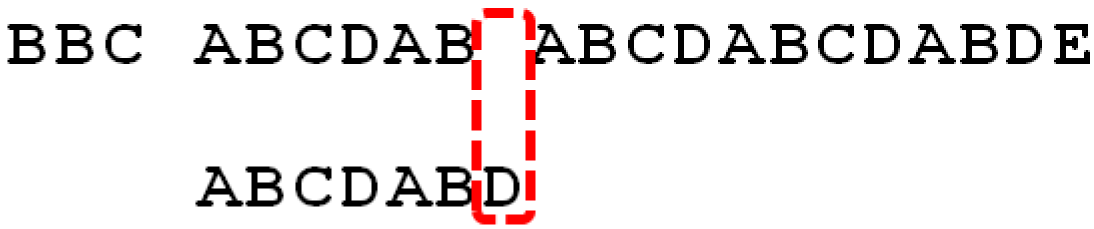
下面，咱们就结合之前的《最大长度表》和上述结论，进行字符串的匹配。如果给定文本串“BBC ABCDAB ABCDAB CDABDE”，和模式串“ABCDABD”，现在要拿模式串去跟文本串匹配，如下图所示：



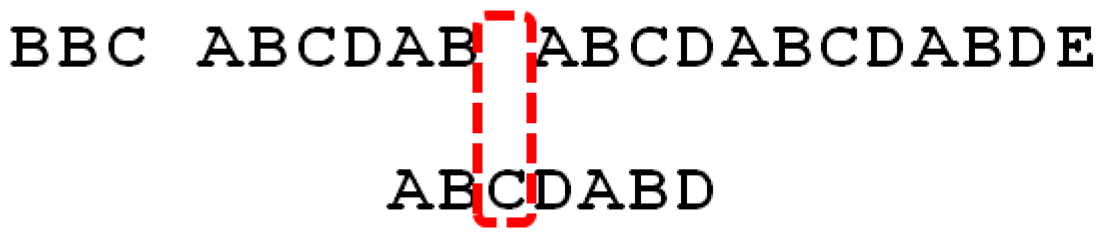
- 1.因为模式串中的字符 A 跟文本串中的字符 B、B、C、空格一开始就不匹配，所以不必考虑结论，直接将模式串不断的右移一位即可，直到模式串中的字符 A 跟文本串的第 5 个字符 A 匹配成功：



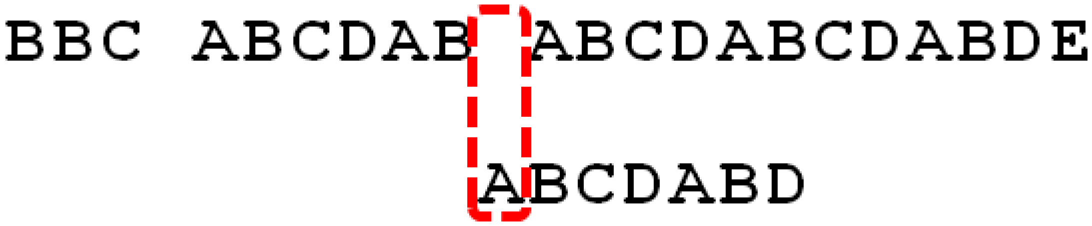
- 2.继续往后匹配，当模式串最后一个字符 D 跟文本串匹配时失配，显而易见，模式串需要向右移动。但向右移动多少位呢？因为此时已经匹配的字符数为 6 个（ABCDAB），然后根据《最大长度表》可得失配字符 D 的上一位字符B对应的长度值为 2，所以根据之前的结论，可知需要向右移动 6 - 2 = 4 位。



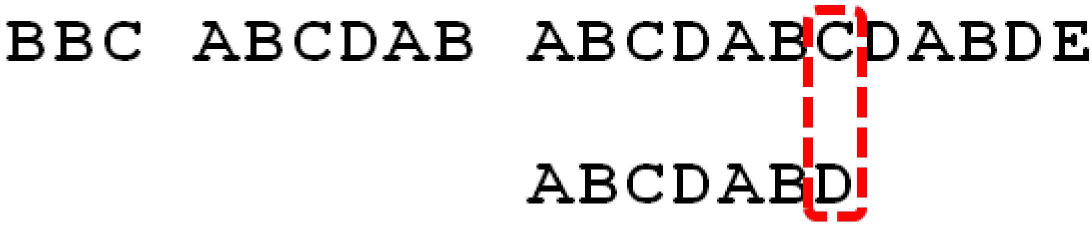
- 3.模式串向右移动 4 位后，发现 C 处再度失配，因为此时已经匹配了 2 个字符（AB），且上一位字符 B 对应的最大长度值为 0，所以向右移动：2 - 0 =2 位。



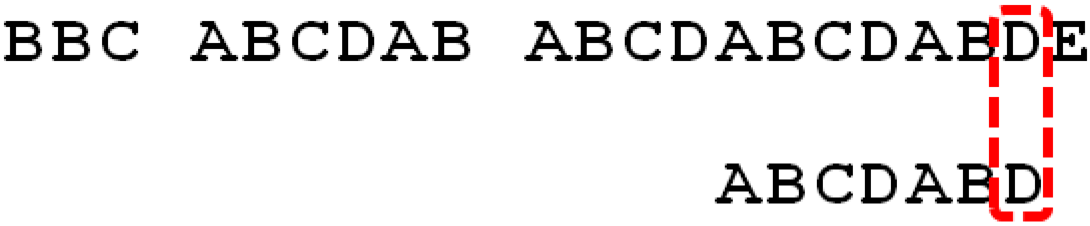
- 4.A 与空格失配，向右移动 1 位。



- 5.继续比较，发现 D 与 C 失配，故向右移动的位数为：已匹配的字符数 6 减去上一位字符 B 对应的最大长度 2，即向右移动 6 - 2 = 4 位。



- 6.经历第 5 步后，发现匹配成功，过程结束。



通过上述匹配过程可以看出，问题的关键就是寻找模式串中最大长度的相同前缀和后缀，找到了模式串中每个字符之前的前缀和后缀公共部分的最大长度后，便可基于此匹配。而这个最大长度便正是 next 数组要表达的含义。

3.3.3 根据《最大长度表》求 next 数组

由上文，我们已经知道，字符串“ABCDABD”各个前缀后缀的最大公共元素长度分别为：

模式串	A	B	C	D	A	B	D
前后缀最大公共元素长度	0	0	0	0	1	2	0

而且，根据这个表可以得出下述结论

- 失配时，模式串向右移动的位数为：已匹配字符数 - 失配字符的上一位字符所对应的最大长度值

上文利用这个表和结论进行匹配时，我们发现，当匹配到一个字符失配时，其实没必要考虑当前失配的字符，更何况我们每次失配时，都是看的失配字符的上一位字符对应的最大长度值。如此，便引出了 next 数组。

给定字符串“ABCDABD”，可求得它的 next 数组如下：

模式串	A	B	C	D	A	B	D
next	-1	0	0	0	0	1	2

把 next 数组跟之前求得的最大长度表对比后，不难发现，next 数组相当于“最大长度值” 整体向右移动一位，然后初始值赋为 -1。意识到了这一点，你会惊呼原来 next 数组的求解竟然如此简单：就是找最大对称长度的前缀后缀，然后整体右移一位，初值赋为 -1（当然，你也可以直接计算某个字符对应的 next 值，就是看这个字符之前的字符串中有多大长度的相同前缀后缀）。

换言之，对于给定的模式串：ABCDABD，它的最大长度表及next 数组分别如下：

模式串	A	B	C	D	A	B	D
最大长度值	0	0	0	0	1	2	0
next 数组	-1	0	0	0	0	1	2

根据最大长度表求出了 next 数组后，从而有

失配时，模式串向右移动的位数为：失配字符所在位置 - 失配字符对应的next 值

而后，你会发现，无论是基于《最大长度表》的匹配，还是基于 next 数组的匹配，两者得出来的向右移动的位数是一样的。为什么呢？因为：

- 根据《最大长度表》，失配时，模式串向右移动的位数 = 已经匹配的字符数 - 失配字符的上一位字符的最大长度值
- 而根据《next 数组》，失配时，模式串向右移动的位数 = 失配字符的位置 - 失配字符对应的 next 值
    - 其中，从 0 开始计数时，失配字符的位置 = 已经匹配的字符数（失配字符不计数），而失配字符对应的 next 值 = 失配字符的上一位字符的最大长度值，两相比较，结果必然完全一致。

所以，你可以把《最大长度表》看做是 next 数组的雏形，甚至就把它当做 next 数组也是可以的，区别不过是怎么用的问题。

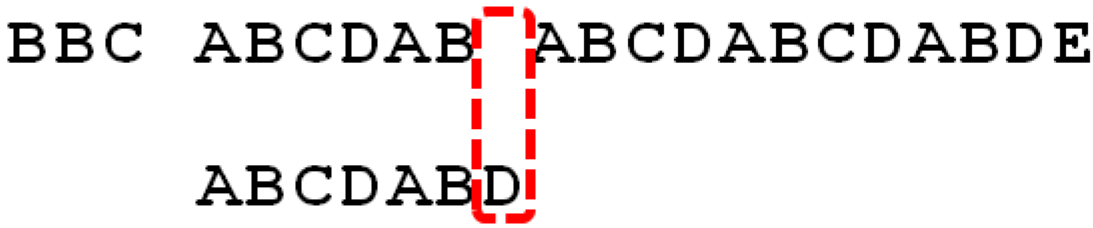
3.3.4 通过代码递推计算 next 数组

接下来，咱们来写代码求下 next 数组。

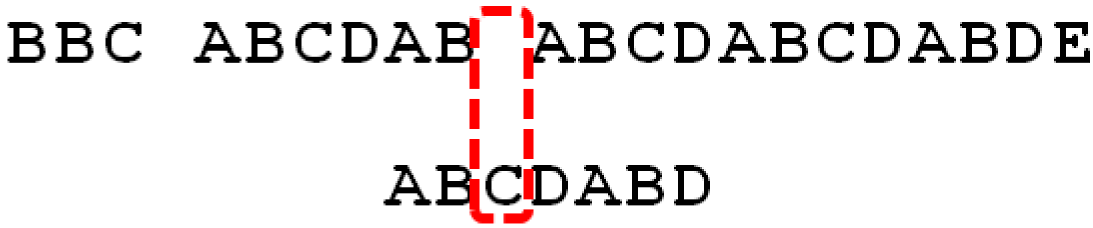
基于之前的理解，可知计算 next 数组的方法可以采用递推：

- 1. 如果对于值 k，已有 p0 p1, ..., pk-1 = pj-k pj-k+1, ..., pj-1，相当于 next[j] = k。 此意味着什么呢？究其本质，**next[j] = k 代表 p[j] 之前的模式串子串中，有长度为 k 的相同前缀和后缀**。有了这个 next 数组，在 KMP 匹配中，当模式串中 j 处的字符失配时，下一步用 next[j] 处的字符继续跟文本串匹配，相当于模式串向右移动 j - next[j] 位。

举个例子，如下图，根据模式串“ABCDABD”的 next 数组可知失配位置的字符 D 对应的 next 值为 2，代表字符 D 前有长度为 2 的相同前缀和后缀（这个相同的前缀后缀即为“AB”），失配后，模式串需要向右移动 j - next [j] = 6 - 2 =4 位。



向右移动 4 位后，模式串中的字符 C 继续跟文本串匹配。



- 2. 下面的问题是：已知 next [0, ..., j]，如何求出 next [j + 1] 呢？

对于 P 的前 j+1 个序列字符：

- 若p[k] == p[j]，则 next[j + 1] = next [j] + 1 = k + 1;
- 若p[k] ≠ p[j]，如果此时 p[next[k]] == p[j]，则 next[j + 1] = next[k] + 1，否则继续递归前缀索引 k = next[k]，而后重复此过程。 相当于在字符 p[j+1] 之前不存在长度为 k+1 的前缀"p0 p1, …, pk-1 pk"跟后缀“pj-k pj-k+1, …, pj-1 pj"相等，那么是否可能存在另一个值 t+1 < k+1，使得长度更小的前缀 “p0 p1, …, pt-1 pt” 等于长度更小的后缀 “pj-t pj-t+1, …, pj-1 pj” 呢？如果存在，那

么这个 t+1 便是 next[j+1] 的值，此相当于利用已经求得的 next 数组（next [0, ..., k, ..., j]）进行 P 串前缀跟 P 串后缀的匹配。

一般的文章或教材可能就此一笔带过，但大部分的初学者可能还是不能很好的理解上述求解 next 数组的原理，故接下来，我再来着重说明下。

如下图所示，假定给定模式串 ABCDABCE，且已知 next [j] = k（相当于“p0 pk-1” = “pj-k pj-1” = AB，可以看出 k 为 2），现要求 next [j + 1] 等于多少？因为 pk = pj = C，所以 next[j + 1] = next[j] + 1 = k + 1（可以看出 next[j + 1] = 3）。代表字符 E 前的模式串中，有长度 k+1 的相同前缀后缀。

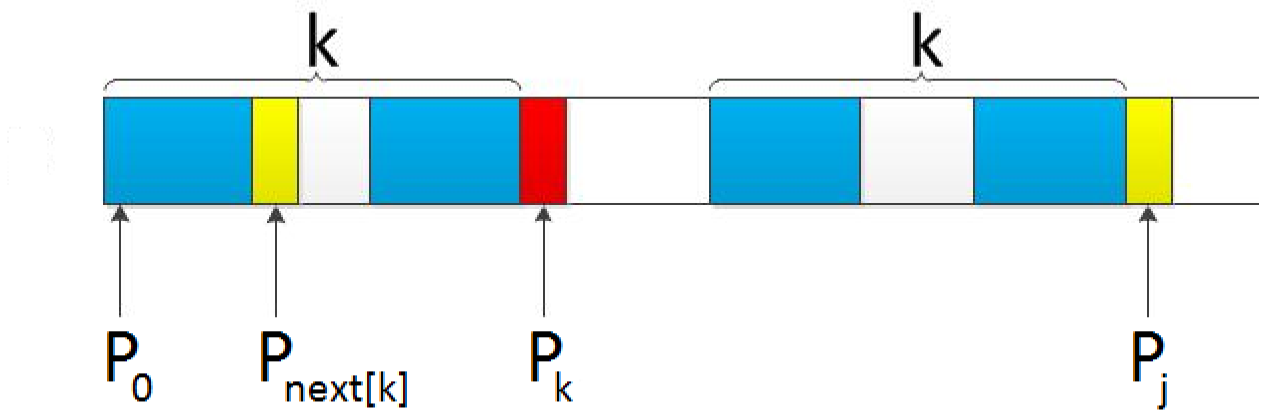
模式串	A	B	C	D	A	B	C	E
前后缀相同长度	0	0	0	0	1	2	3	0
next 值	-1	0	0	0	0	1	2	?
索引	p_0	p_{k-1}	p_k	p_{k+1}	p_{j-k}	p_{j-1}	p_j	p_{j+1}

但如果 pk != pj 呢？说明“p0 pk-1 pk” ≠ “pj-k pj-1 pj”。换言之，当 pk != pj 后，字符 E 前有多大长度的相同前缀后缀呢？很明显，因为 C 不同于 D，所以 ABC 跟 ABD 不相同，即字符 E 前的模式串没有长度为 k+1 的相同前缀后缀，也就不能再简单的令：next[j + 1] = next[j] + 1 。所以，咱们只能去寻找长度更短一点的相同前缀后缀。

模式串	A	B	C	D	A	B	D	E
前后缀相同长度	0	0	0	0	1	2	0	0
next 值	-1	0	0	0	0	1	2	?
索引	p_0	p_{k-1}	p_k	p_{k+1}	p_{j-k}	p_{j-1}	p_j	p_{j+1}

结合上图来讲，若能在前缀“ p0 pk-1 pk ” 中不断的递归前缀索引 k = next [k]，找到一个字符 pk’ 也为 D，代表 pk’ = pj，且满足 p0 pk’-1 pk’ = pj-k’ pj-1 pj，则最大相同的前缀后缀长度为 k’ + 1，从而 next [j + 1] = k’ + 1 = next [k’] + 1。否则前缀中没有 D，则代表没有相同的前缀后缀，next [j + 1] = 0。

那为何递归前缀索引k = next[k]，就能找到长度更短的相同前缀后缀呢？这又归根到 next 数组的含义。我们拿前缀 p0 pk-1 pk 去跟后缀 pj-k pj-1 pj 匹配，如果 pk 跟 pj 失配，下一步就是用 p[next[k]] 去跟 pj 继续匹配，如果 p[next[k]]跟 pj 还是不匹配，则需要寻找长度更短的相同前缀后缀，即下一步用 p[next[next[k]]] 去跟 pj 匹配。此过程相当于模式串的自我匹配，所以不断的递归 k = next[k]，直到要么找到长度更短的相同前缀后缀，要么没有长度更短的相同前缀后缀。如下图所示：



所以，因最终在前缀 ABC 中没有找到 D，故 E 的 next 值为 0：

模式串的后缀：ABDE

模式串的前缀：ABC

前缀右移两位：　　ABC

读到此，有的读者可能又有疑问了，那能否举一个能在前缀中找到字符 D 的例子呢？OK，咱们便来看一个能在前缀中找到字符 D 的例子，如下图所示：

模式串	D	A	B	C	D	A	B	D	E
最长相同前缀后缀	0	0	0	0	1	2	3	?	
next 值	-1	0	0	0	0	1	2	3	?
索引	p_0	p_1	p_{k-1}	p_k	p_{j-k}	p_{j-2}	p_{j-1}	p_j	p_{j+1}

给定模式串 DABCDABDE，我们很顺利的求得字符 D 之前的“DABCDAB”的各个子串的最长相同前缀后缀的长度分别为 0 0 0 0 1 2 3，但当遍历到字符 D，要求包括 D 在内的“DABCDABD”最长相同前缀后缀时，我们发现 pj 处的字符 D 跟 pk 处的字符 C 不一样，换言之，前缀 DABC 的最后一个字符 C 跟后缀 DABD 的最后一个字符 D 不相同，所以不存在长度为 4 的相同前缀后缀。

怎么办呢？既然没有长度为 4 的相同前缀后缀，咱们可以寻找长度短点的相同前缀后缀，最终，因在 p0 处发现也有个字符 D，p0 = pj，所以 p[j] 对应的长度值为 1，相当于 E 对应的 next 值为 1（即字符 E 之前的字符串“DABCDABD”中有长度为 1 的相同前缀和后缀）。

综上，可以通过递推求得 next 数组，代码如下所示：

```
void GetNext(char* p,int next[])
{
    int pLen = strlen(p);
    next[0] = -1;
    int k = -1;
    int j = 0;
    while (j < pLen - 1)
    {
        //p[k]表示前缀，p[j]表示后缀
        if (k == -1 || p[j] == p[k])
        {
            ++k;
            ++j;
            next[j] = k;
        }
        else
        {
            k = next[k];
        }
    }
}
```

用代码重新计算下“ABCDABD”的 next 数组，以验证之前通过“最长相同前缀后缀长度值右移一位，然后初值赋为 -1” 得到的 next 数组是否正确，计算结果如下表格所示：

模式串	A	B	C	D	A	B	D
k	-1	0	-1,0	-1,0	-1,0	1	2
j	0	1	2	3	4	5	6
next 数组	-1	0	0	0	0	1	2

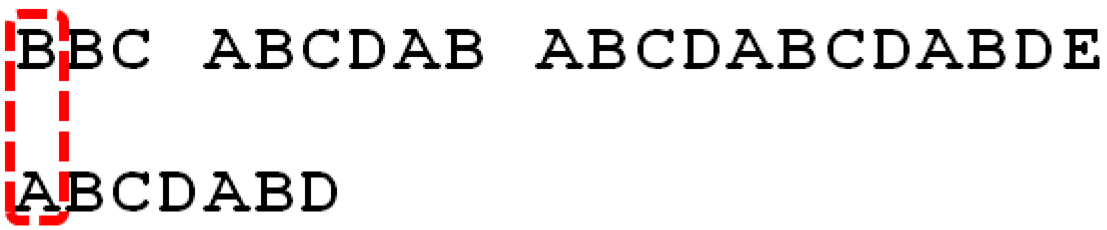
从上述表格可以看出，无论是之前通过“最长相同前缀后缀长度值右移一位，然后初值赋为 -1” 得到的 next 数组，还是之后通过代码递推计算求得的 next 数组，结果是完全一致的。

3.3.5 基于《next 数组》匹配

下面，我们来基于 next 数组进行匹配。

字符	A	B	C	D	A	B	D
Next值	-1	0	0	0	0	1	2

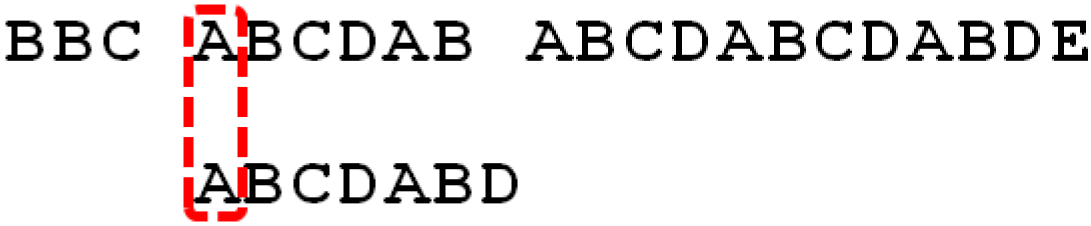
还是给定文本串“BBC ABCDAB ABCDABCDABDE”，和模式串“ABCDABD”，现在要拿模式串去跟文本串匹配，如下图所示：



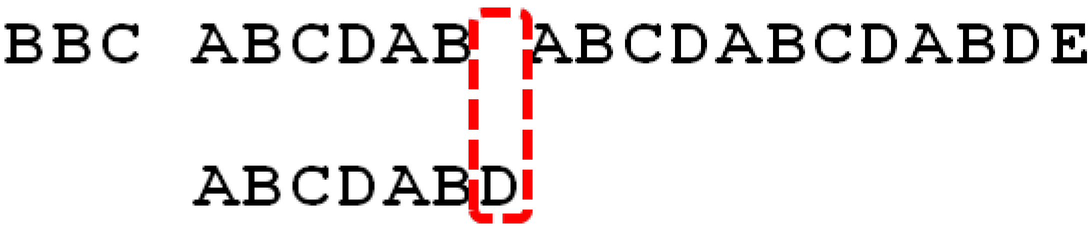
在正式匹配之前，让我们来再次回顾下上文 2.1 节所述的 KMP 算法的匹配流程：

- “假设现在文本串S匹配到 i 位置，模式串 P 匹配到 j 位置
  - 如果 j = -1，或者当前字符匹配成功（即 S[i] == P[j]），都令 i++，j++，继续匹配下一个字符；
  - 如果 j != -1，且当前字符匹配失败（即 S[i] != P[j]），则令 i 不变，j = next[j]。此举意味着失配时，模式串 P 相对于文本串 S 向右移动了 j - next [j] 位。
    - 换言之，当匹配失败时，模式串向右移动的位数为：失配字符所在位置 - 失配字符对应的 next 值，即移动的实际位数为：j - next[j]，且此值大于等于1。”
- 1.最开始匹配时
  - P[0] 跟 S[0] 匹配失败
  - 所以执行“如果 j != -1，且当前字符匹配失败（即 S[i] != P[j]），则令 i 不变，j = next[j]”，所以 j = -1，故转而执行“如果 j = -1，或者当前字符匹配成功（即 S[i] == P[j]），都令 i++，j++”，得到i = 1，j = 0，即 P[0] 继续跟 S[1] 匹配。
  - P[0] 跟 S[1] 又失配，j 再次等于 -1，i、j 继续自增，从而 P[0] 跟 S[2] 匹配。
  - P[0] 跟 S[2] 失配后，P[0] 又跟 S[3] 匹配。

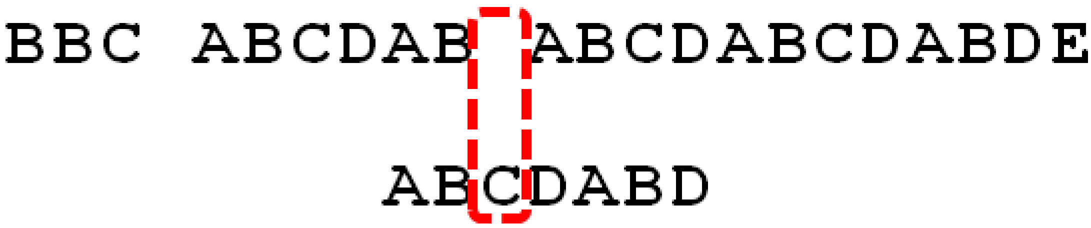
    - P[0] 跟 S[3] 再失配，直到 P[0] 跟 S[4] 匹配成功，开始执行此条指令的后半段：“如果 j = -1，或者当前字符匹配成功（即 S[i] == P[j]），都令 i++，j++”。



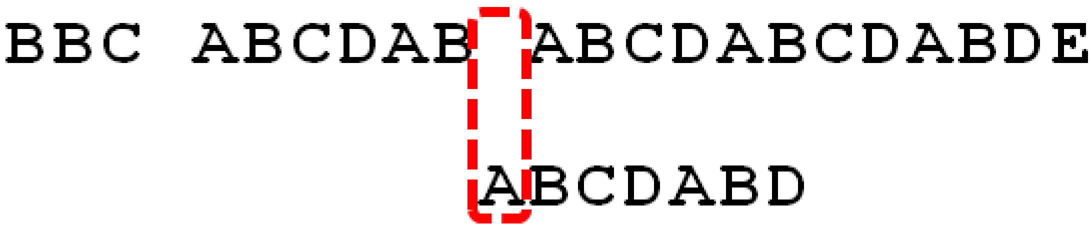
- 2.P[1] 跟 S[5] 匹配成功，P[2] 跟 S[6] 也匹配成功，...，直到当匹配到 P[6] 处的字符 D 时失配（即 S[10] != P[6]），由于 P[6] 处的 D 对应的 next 值为 2，所以下一步用 P[2] 处的字符 C 继续跟 S[10] 匹配，相当于向右移动：j - next[j] = 6 - 2 =4 位。



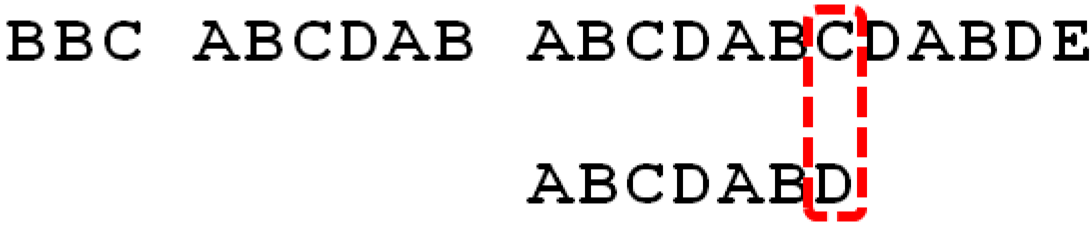
- 3.向右移动 4 位后，P[2] 处的 C 再次失配，由于 C 对应的 next 值为 0，所以下一步用 P[0] 处的字符继续跟 S[10] 匹配，相当于向右移动：j - next[j] = 2 - 0 = 2 位。



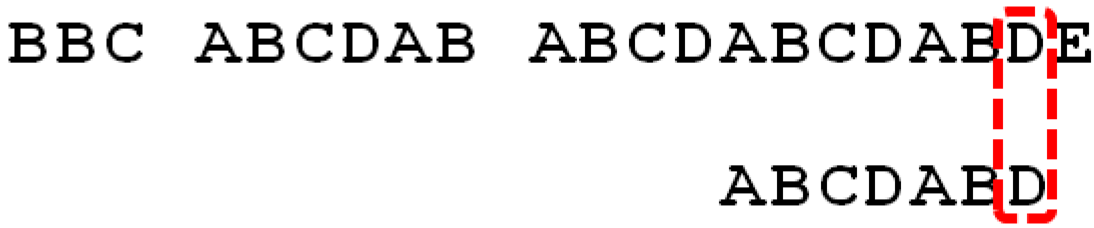
- 4.移动两位之后，A 跟空格不匹配，模式串后移 1 位。



- 5.P[6] 处的 D 再次失配，因为 P[6] 对应的 next 值为 2，故下一步用 P[2] 继续跟文本串匹配，相当于模式串向右移动 j - next[j] = 6 - 2 = 4 位。



- 6.匹配成功，过程结束。



匹配过程一模一样。也从侧面佐证了，next 数组确实是只要将各个最大前缀后缀的公共元素的长度值右移一位，且把初值赋为 -1 即可。

3.3.6 基于《最大长度表》与基于《next 数组》等价

我们已经知道，利用 next 数组进行匹配失配时，模式串向右移动 j - next [j] 位，等价于已匹配字符数 - 失配字符的上一位字符所对应的最大长度值。原因是：

- j 从 0 开始计数，那么当数到失配字符时，j 的数值就是已匹配的字符数；
- 由于 next 数组是由最大长度值表整体向右移动一位（且初值赋为 -1）得到的，那么失配字符的上一位字符所对应的最大长度值，即为当前失配字符的 next 值。

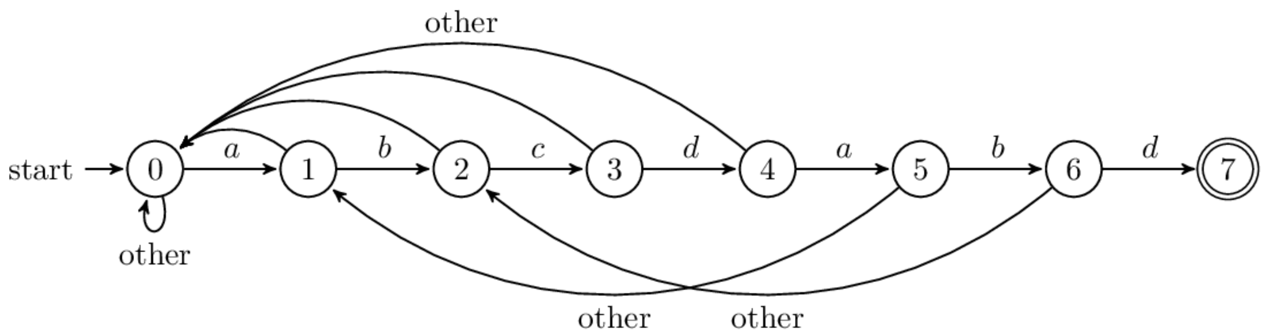
但为何本文不直接利用 next 数组进行匹配呢？因为 next 数组不好求，而一个字符串的前缀后缀的公共元素的最大长度值很容易求。例如若给定模式串“ababa”，要你快速口算出其 next 数组，乍一看，每次求对应字符的 next 值时，还得把该字符排除之外，然后看该字符之前的字符串中有最大长度为多大的相同前缀后缀，此过程不够直接。而如果让你求其前缀后缀公共元素的最大长度，则很容易直接得出结果：0 0 1 2 3，如下表格所示：

模式串的各个子串	前缀	后缀	最大公共元素长度
a	空	空	0
ab	a	b	0
aba	a,ab	a,ba	1
abab	a,ab,aba	b,ab,bab	2
ababa	a,ab,aba,abab	a,ba,aba,baba	3

然后这 5 个数字 全部整体右移一位，且初值赋为 -1，即得到其 next 数组：-1 0 0 1 2。

3.3.7 Next 数组与有限状态自动机

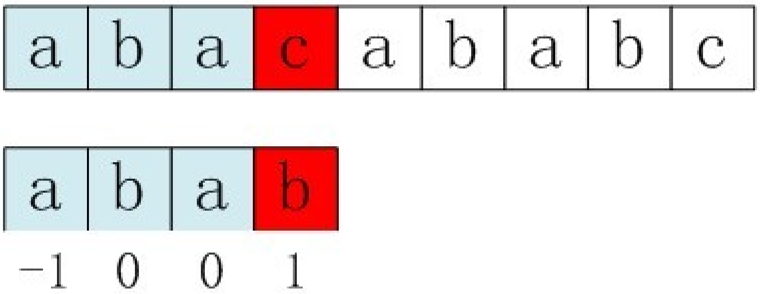
next 负责把模式串向前移动，且当第j位不匹配的时候，用第 next[j] 位和主串匹配，就像打了张“表”。此外，next 也可以看作有限状态自动机的状态，在已经读了多少字符的情况下，失配后，前面读的若干个字符是有用的。



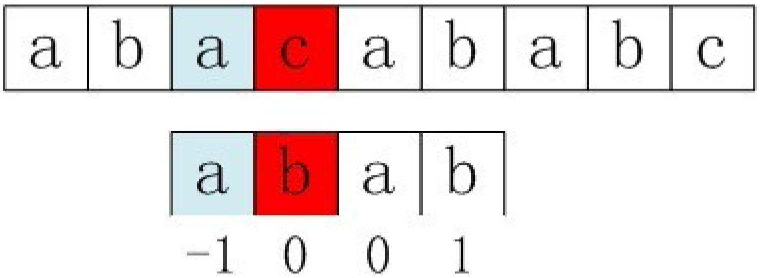
3.3.8 Next 数组的优化

行文至此，咱们全面了解了暴力匹配的思路、KMP算法的原理、流程、流程之间的内在逻辑联系，以及 next 数组的简单求解（《最大长度表》整体右移一位，然后初值赋为 -1）和代码求解，最后基于《next 数组》的匹配，看似洋洋洒洒，清晰透彻，但以上忽略了一个小问题。

比如，如果用之前的 next 数组方法求模式串“abab”的 next 数组，可得其 next 数组为-1 0 0 1（0 0 1 2整体右移一位，初值赋为 -1），当它跟下图中的文本串去匹配的时候，发现 b 跟 c 失配，于是模式串右移 j - next[j] = 3 - 1 =2 位。



右移 2 位后，b 又跟 c 失配。事实上，因为在上一步的匹配中，已经得知 p[3] = b，与 s[3] = c 失配，而右移两位之后，让 p[next[3]] = p[1] = b 再跟 s[3] 匹配时，必然失配。问题出在哪呢？



问题出在不该出现 p[j] = p[next[j]]。为什么呢？理由是：当 p[j] != s[i] 时，下次匹配必然是 p[next[j]] 跟 s[i] 匹配，如果 p[j] = p[next[j]]，必然导致后一步匹配失败（因为 p[j] 已经跟 s[i] 失配，然后你还用跟 p[j] 等同的值 p[next[j]] 去跟 s[i] 匹配，很显然，必然失配），所以不能允许 p[j] = p[next[j]]。如果出现了 p[j] = p[next[j]] 咋办呢？如果出现了，则需要再次递归，即令 next[j] = next[next[j]]。

所以，咱们得修改下求 next 数组的代码。

```
//优化过后的next 数组求法
void GetNextval(char* p, int next[])
{
    int pLen = strlen(p);
    next[0] = -1;
    int k = -1;
    int j = 0;
    while (j < pLen - 1)
    {
        //p[k]表示前缀，p[j]表示后缀
        if (k == -1 || p[j] == p[k])
        {
            ++j;
            ++k;
```

```
            //较之前next数组求法，改动在下面4行
            if (p[j] != p[k])
                next[j] = k;   //之前只有这一行
            else
                //因为不能出现p[j] = p[ next[j ]]，所以当出现时需要继续递归，k = next[k] = next[next[k]]
                next[j] = next[k];
        }
        else
        {
            k = next[k];
        }
    }
}
```

利用优化过后的 next 数组求法，可知模式串“abab”的新 next 数组为：-1 0 -1 0。可能有些读者会问：原始 next 数组是前缀后缀最长公共元素长度值右移一位， 然后初值赋为 -1 而得，那么优化后的 next 数组如何快速心算出呢？实际上，只要求出了原始 next 数组，便可以根据原始 next 数组快速求出优化后的 next 数组。还是以 abab 为例，如下表格所示：

模式串	a	b	a	b
最大长度值	0	0	1	2
未优化next数组	next[0] = -1	next[1] = 0	next[2] = 0	next[3] = 1
索引值	p_0	p_1	p_2	p_3
优化理由	初值不变	p[1] !=p[next[1]]	因pj不能等于p[next[j]]，即p[2]不能等于p[next[2]]	p[3]不能等于p[next[3]]
措施	无需处理	无需处理	next[2]=next[next[2]]=next[0]=-1	next[3]=next[next[3]]=next[1]=0
优化的next数组	-1	0	-1	0

只要出现了 p[next[j]] = p[j] 的情况，则把 next[j] 的值再次递归。例如在求模式串“abab”的第 2 个 a 的 next 值时，如果是未优化的 next 值的话，第 2 个 a 对应的 next 值为 0，相当于第 2 个 a 失配时，下一步匹配模式串会用 p[0] 处的 a 再次跟文本串匹配，必然失配。所以求第 2 个 a 的 next 值时，需要再次递归：next[2] = next[next[2]] = next[0] = -1（此后，根据优化后的新 next 值可知，第 2 个 a 失配时，执行“如果 j = -1，或者当前字符匹配成功（即 S[i] == P[j]），都令 i++，j++，继续匹配下一个字符”），同理，第 2 个 b 对应的 next 值为 0。

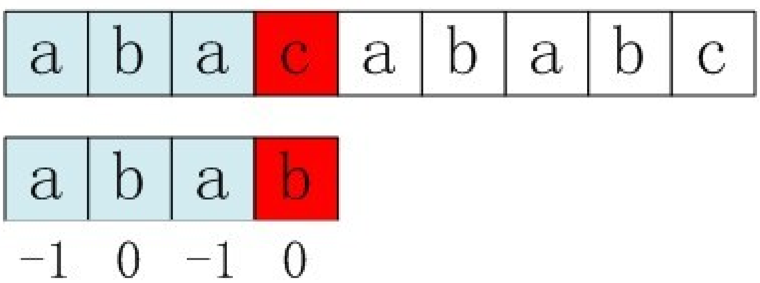
对于优化后的 next 数组可以发现一点：如果模式串的后缀跟前缀相同，那么它们的 next 值也是相同的，例如模式串 abcabc，它的前缀后缀都是 abc，其优化后的 next 数组为：-1 0 0 -1 0 0，前缀后缀 abc 的 next 值都为 -1 0 0。

然后引用下之前 3.1 节的 KMP 代码：

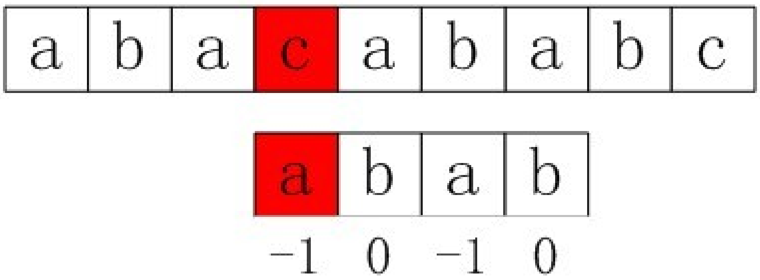
```
int KmpSearch(char* s, char* p)
{
    int i = 0;
    int j = 0;
    int sLen = strlen(s);
    int pLen = strlen(p);
    while (i < sLen && j < pLen)
    {
        //①如果j = -1，或者当前字符匹配成功（即S[i] == P[j]），都令i++，j++
        if (j == -1 || s[i] == p[j])
        {
            i++;
            j++;
        }
        else
        {
            //②如果j != -1，且当前字符匹配失败（即S[i] != P[j]），则令 i 不变，j = next[j]
            //next[j]即为j所对应的next值
            j = next[j];
        }
    }
    if (j == pLen)
        return i - j;
    else
        return -1;
}
```

接下来，咱们继续拿之前的例子说明，整个匹配过程如下：

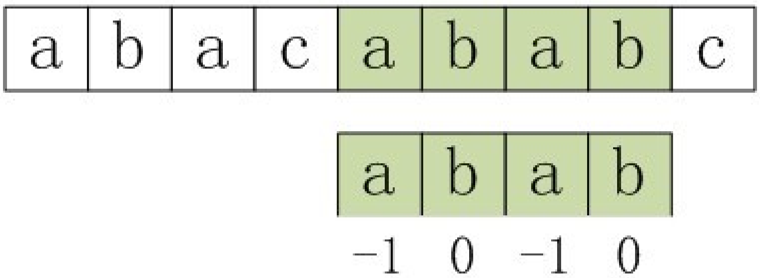
1.S[3] 与 P[3] 匹配失败。



2.S[3] 保持不变，P 的下一个匹配位置是 P[next[3]]，而 next[3]=0，所以 P[next[3]]=P[0] 与 S[3] 匹配。



3.由于上一步骤中 P[0] 与 S[3] 还是不匹配。此时 i=3，j=next [0]=-1，由于满足条件 j==-1，所以执行“++i，++j”，即主串指针下移一个位置，P[0] 与 S[4] 开始匹配。最后 j==pLen，跳出循环，输出结果 i - j = 4（即模式串第一次在文本串中出现的位置），匹配成功，算法结束。



3.4 KMP 的时间复杂度分析

相信大部分读者读完上文之后，已经发觉其实理解 KMP 非常容易，无非是循序渐进把握好下面几点：

- 如果模式串中存在相同前缀和后缀，即 pj-k pj-k+1, ..., pj-1 = p0 p1, ..., pk-1，那么在 pj 跟 si 失配后，让模式串的前缀 p0 p1...pk-1 对应着文本串 si-k si-k+1...si-1，而后让 pk 跟 si 继续匹配。
- 之前本应是 pj 跟 si 匹配，结果失配了，失配后，令 pk 跟 si 匹配，相当于 j 变成了 k，模式串向右移动 j - k 位。
- 因为 k 的值是可变的，所以我们用 next[j] 表示j处字符失配后，下一次匹配模式串应该跳到的位置。换言之，失配前是 j，pj 跟 si 失配时，用 p[next[j]]继续跟 si 匹配，相当于 j 变成了 next[j]，所以，j = next[j]，等价于把模式串向右移动 j - next [j] 位。
- 而 next[j] 应该等于多少呢？ next[j] 的值由 j 之前的模式串子串中有多大长度的相同前缀后缀所决定，如果 j 之前的模式串子串中（不含 j）有最大长度为k的相同前缀后缀，那么 next [j] = k。

如之前的图所示：

接下来，咱们来分析下 KMP 的时间复杂度。分析之前，先来回顾下 KMP 匹配算法的流程：

“KMP 的算法流程：

- 假设现在文本串 S 匹配到 i 位置，模式串 P 匹配到 j 位置
  - 如果 j = -1，或者当前字符匹配成功（即 S[i] == P[j]），都令 i++，j++，继续匹配下一个字符；
  - 如果 j != -1，且当前字符匹配失败（即 S[i] != P[j]），则令 i 不变，j = next[j]。此举意味着失配时，模式串 P 相对于文本串 S 向右移动了 j - next [j] 位。”

我们发现如果某个字符匹配成功，模式串首字符的位置保持不动，仅仅是 i++、j++；如果匹配失配，i 不变（即 i 不回溯），模式串会跳过匹配过的 next [j] 个字符。整个算法最坏的情况是，当模式串首字符位于 i - j 的位置时才匹配成功，算法结束。

所以，如果文本串的长度为 n，模式串的长度为 m，那么匹配过程的时间复杂度为 O(n)，算上计算 next 的 O(m) 时间，KMP 的整体时间复杂度为 O(m + n)。

4

4. 扩展 1：BM 算法

KMP 的匹配是从模式串的开头开始匹配的，而 1977 年，德克萨斯大学的 Robert S. Boyer 教授和 J Strother Moore 教授发明了一种新的字符串匹配算法：Boyer-Moore 算法，简称 BM 算法。该算法从模式串的尾部开始匹配，且拥有在最坏情况下 O(N) 的时间复杂度。在实践中，比 KMP 算法的实际效能高。

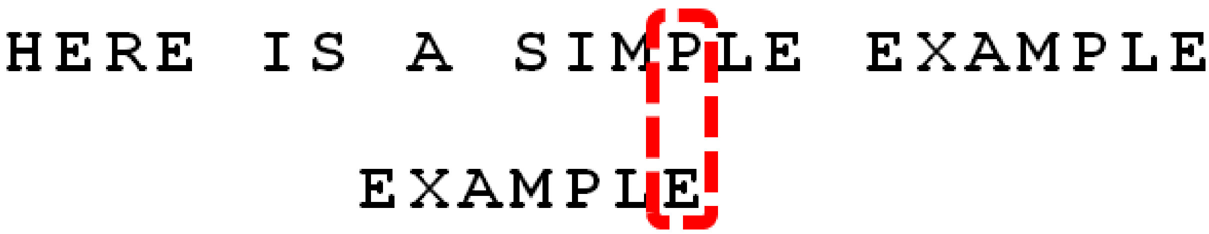
BM 算法定义了两个规则：

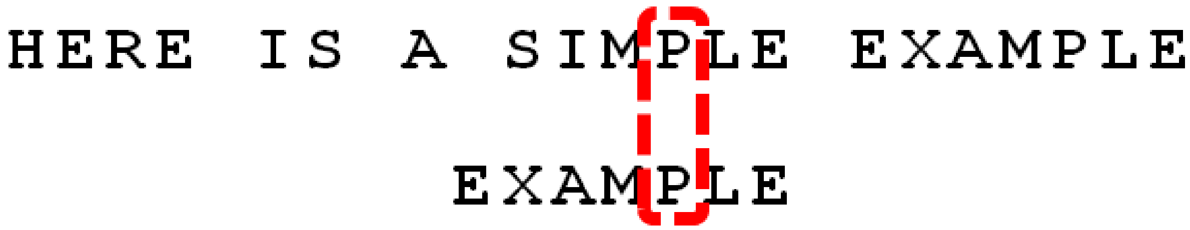
- 坏字符规则：当文本串中的某个字符跟模式串的某个字符不匹配时，我们称文本串中的这个失配字符为坏字符，此时模式串需要向右移动，移动的位数 = 坏字符在模式串中的位置 - 坏字符在模式串中最右出现的位置。此外，如果"坏字符"不包含在模式串之中，则最右出现位置为 -1。
- 好后缀规则：当字符失配时，后移位数 = 好后缀在模式串中的位置 - 好后缀在模式串上一次出现的位置，且如果好后缀在模式串中没有再次出现，则为 -1。

下面举例说明 BM 算法。例如，给定文本串“HERE IS A SIMPLE EXAMPLE”，和模式串“EXAMPLE”，现要查找模式串是否在文本串中，如果存在，返回模式串在文本串中的位置。

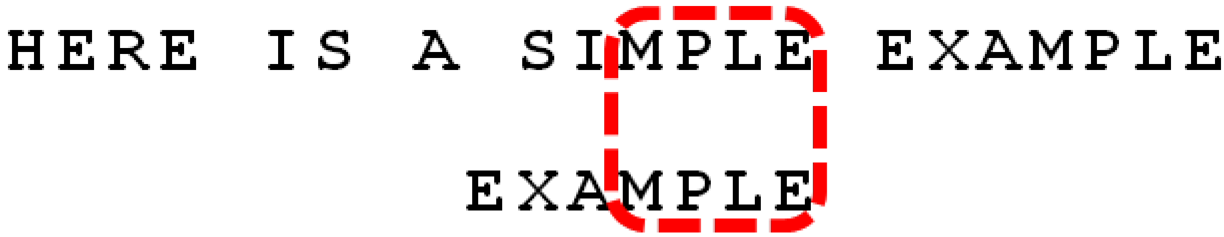
1.首先，"文本串"与"模式串"头部对齐，从尾部开始比较。"S"与"E"不匹配。这时，"S"就被称为"坏字符"（bad character），即不匹配的字符，它对应着模式串的第 6 位。且"S"不包含在模式串"EXAMPLE"之中（相当于最右出现位置是 -1），这意味着可以把模式串后移 6-(-1)=7 位，从而直接移到"S"的后一位。

2.依然从尾部开始比较，发现"P"与"E"不匹配，所以"P"是"坏字符"。但是，"P"包含在模式串"EXAMPLE"之中。因为“P”这个“坏字符”对应着模式串的第 6 位（从 0 开始编号），且在模式串中的最右出现位置为 4，所以，将模式串后移 6-4=2 位，两个"P"对齐。

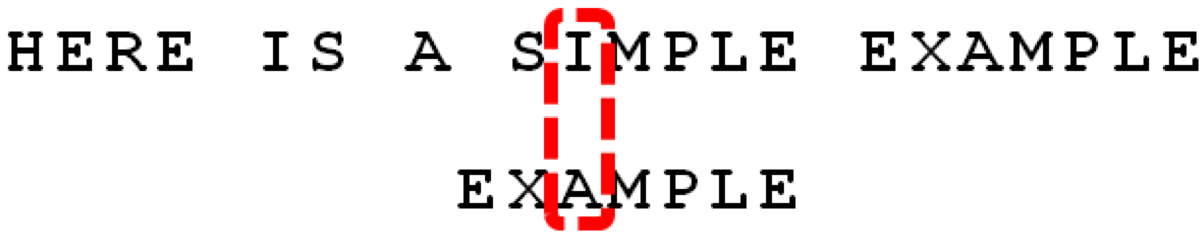


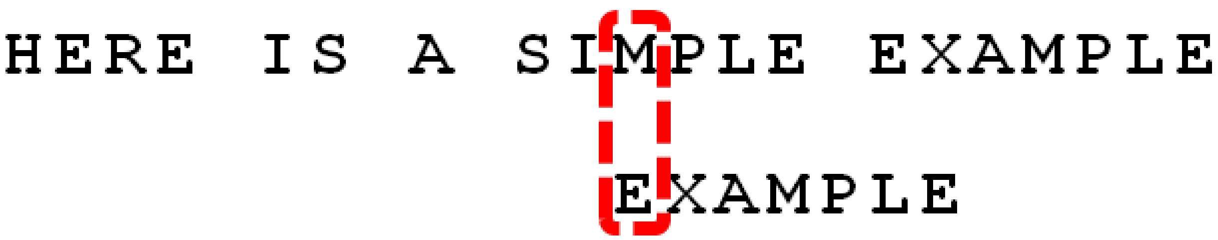


3.依次比较，得到 “MPLE”匹配，称为"好后缀"（good suffix），即所有尾部匹配的字符串。注意，"MPLE"、"PLE"、"LE"、"E"都是好后缀。



4.发现“I”与“A”不匹配：“I”是坏字符。如果是根据坏字符规则，此时模式串应该后移 2-(-1)=3 位。问题是，有没有更优的移法？

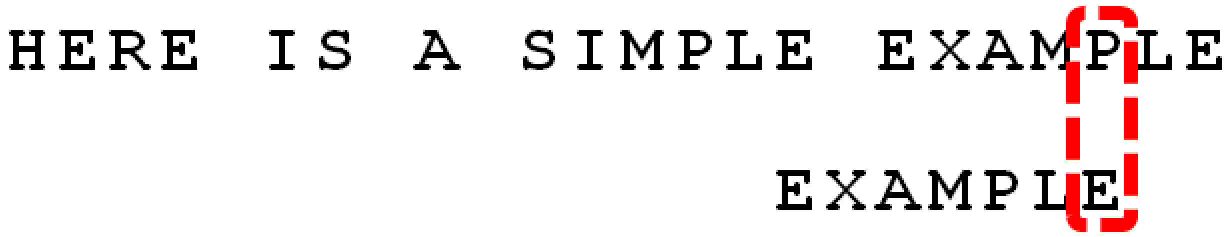




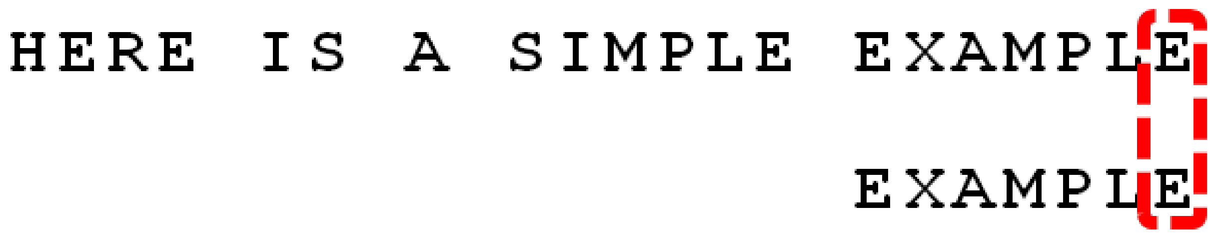
5.更优的移法是利用好后缀规则：当字符失配时，后移位数 = 好后缀在模式串中的位置 - 好后缀在模式串中上一次出现的位置，且如果好后缀在模式串中没有再次出现，则为 -1。

所有的“好后缀”（MPLE、PLE、LE、E）之中，只有“E”在“EXAMPLE”的头部出现，所以后移 6-0=6 位。

可以看出，“坏字符规则”只能移3位，“好后缀规则”可以移 6 位。每次后移这两个规则之中的较大值。这两个规则的移动位数，只与模式串有关，与原文本串无关。



6.继续从尾部开始比较，“P”与“E”不匹配，因此“P”是“坏字符”，根据“坏字符规则”，后移 6 - 4 = 2 位。因为是最后一位就失配，尚未获得好后缀。



由上可知，BM 算法不仅效率高，而且构思巧妙，容易理解。

5

5. 扩展 2：Sunday 算法

上文中，我们已经介绍了 KMP 算法和 BM 算法，这两个算法在最坏情况下均具有线性的查找时间。但实际上，KMP 算法并不比最简单的 c 库函数 strstr() 快多少，而 BM 算法虽然通常比 KMP 算法快，但 BM 算法也还不是现有字符串查找算法中最快的算法，本文最后再介绍一种比 BM 算法更快的查找算法即 Sunday 算法。

Sunday 算法由 Daniel M.Sunday 在 1990 年提出，它的思想跟 BM 算法很相似：

- 只不过 Sunday 算法是从前往后匹配，在匹配失败时关注的是文本串中参加匹配的最末位字符的下一位字符。
    - 如果该字符没有在模式串中出现则直接跳过，即移动位数 = 匹配串长度 + 1；
    - 否则，其移动位数 = 模式串中最右端的该字符到末尾的距离 +1。

下面举个例子说明下 Sunday 算法。假定现在要在文本串"substring searching algorithm"中查找模式串"search"。

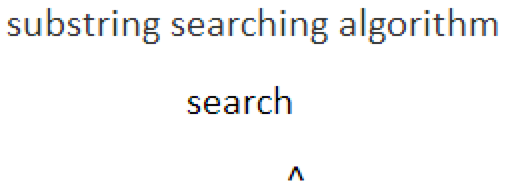
1.刚开始时，把模式串与文本串左边对齐：

```
substring searching algorithm
search
^
```

2.结果发现在第 2 个字符处发现不匹配，不匹配时关注文本串中参加匹配的最末位字符的下一位字符，即标粗的字符 i，因为模式串 search 中并不存在 i，所以模式串直接跳过一大片，向右移动位数 = 匹配串长度 + 1 = 6 + 1 = 7，从 i 之后的那个字符（即字符 n）开始下一步的匹配，如下图：

```
substring searching algorithm
          search
            ^
```

3.结果第一个字符就不匹配，再看文本串中参加匹配的最末位字符的下一位字符，是'r'，它出现在模式串中的倒数第3位，于是把模式串向右移动 3 位（r 到模式串末尾的距离 + 1 = 2 + 1 =3），使两个'r'对齐，如下：



4. 匹配成功。

回顾整个过程，我们只移动了两次模式串就找到了匹配位置，缘于 Sunday 算法每一步的移动量都比较大，效率很高。完。

6

6. 参考文献

1. 《算法导论》的第十二章：字符串匹配；
2. 本文中模式串“ABCDABD”的部分图来自于此文：http://www.ruanyifeng.com/blog/2013/05/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm.html；
3. 本文 3.3.7 节中有限状态自动机的图由微博网友@龚陆安 绘制：http://d.pr/i/NEiz；
4. 北京 7 月暑假班邹博半小时KMP视频：http://www.julyedu.com/video/play/id/5；
5. 北京 7 月暑假班邹博第二次课的PPT：http://yun.baidu.com/s/1mgFmw7u；
6. 理解 KMP 的 9 张 PPT：http://weibo.com/1580904460/BeCCYrKz3#_rnd1405957424876；
7. 详解 KMP 算法（多图）：http://www.cnblogs.com/yjiyjige/p/3263858.html；
8. 本文第4部分的BM算法参考自此文：http://www.ruanyifeng.com/blog/2013/05/boyer-moore_string_search_algorithm.html；
9. http://youlvconglin.blog.163.com/blog/static/5232042010530101020857；
10. 《数据结构 第二版》，严蔚敏 & 吴伟民编著；
11. http://blog.csdn.net/v_JULY_v/article/details/6545192；
12. http://blog.csdn.net/v_JULY_v/article/details/6111565；
13. Sunday 算法的原理与实现：http://blog.chinaunix.net/uid-22237530-id-1781825.html；
14. 模式匹配之 Sunday 算法：http://blog.csdn.net/sunnianzhong/article/details/8820123；
15. 一篇 KMP 的英文介绍：http://www.inf.fh-flensburg.de/lang/algorithmen/pattern/kmpen.htm；
16. 我 2014 年 9 月 3 日在西安电子科技大学的面试&算法讲座视频（第 36 分钟~第 94 分钟讲 KMP）：http://www.julyedu.com/video/play/id/7。
17. 一幅图理解 KMP next 数组的求法：http://www.rudy-yuan.net/archives/182/。

7

7. 后记

对之前混乱的文章给广大读者带来的困扰表示致歉，对重新写就后的本文即将给读者带来的清晰表示欣慰。希望大部分的初学者，甚至少部分的非计算机专业读者也能看懂此文。有任何问题，欢迎随时批评指正，thanks。

July、二零一四年八月二十二日晚九点。