

结构之法 算法之道

面试、算法、机器学习在线课程：julyedu.com

☰ 目录视图

摘要视图

RSS 订阅

个人资料



v JULY v



访问：12967406次

积分： 49878

等级： **BLOC** 8

排名： 第63名

原创： 158篇 转载： 0篇

译文： 6篇 评论： 14877
条

博主简介

July, 于2010年10月11日开始在CSDN上写博客(搜索:“结构之法”,进入本博客),博客专注面试、算法、机器学习。2015年正式创业,七月在线创始人兼CEO,公司官网:七月在线(<https://www.julyedu.com/>),微博@研究者July。新书《编程之法》15年10月14日起正式上市。JulyEdu c++/算法Q群:123531805, July, 2016/5/6。

July和他朋友们的创业平台



我的微博

[置顶] 从头到尾彻底理解KMP (2014年8月22日版)

标签： 算法 function string delete 数据结构

2011-12-05 13:05

285898人阅读

[评论\(396\)](#)

收藏 举报

分类: 02.Algorithms (后续) (21) 01.Algorithms (研究) (26)

目录(?)

[+]

从头到尾彻底理解KMP

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。后收录于**新书**《编程之法：面试和算法心得》第4.4节中。

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。

然近期因开了个**算法班**，班上专门讲解数据结构、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、以及算法班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. 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 $S[i] \neq P[j]$ ），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])
        {
            //如果当前字符匹配成功（即S[i] == P[j]），则i++，j++
            i++;
            j++;
        }
        else
        {

```



研究者July

加关注

[汗] //@飞行的海螺: @研究者July 像不像你[阴险]

视觉机器人 : 感受一下小李子的洗脑大法,《华尔街之狼》中一个精彩片段,希望努力工作的你,有个清醒的头脑啊。
<http://t.cn/R0X3IL1>

July新书《编程之法》上市



京东 当当 天猫 Amazon
异步社区 互动出版网

文章分类

- 03.Algorithms（实现）（9）
- 01.Algorithms（研究）（27）
- 02.Algorithms（后续）（22）
- 04.Algorithms（讨论）（1）
- 05.MS 100' original（7）
- 06.MS 100' answers（13）
- 07.MS 100' classify（4）
- 08.MS 100' one Keys（6）
- 09.MS 100' follow-up（4）
- 10.MS 100' comments（4）
- 11.TAOPP（编程艺术）（34）
- 12.TAOPP string（8）
- 13.TAOPP array（14）
- 14.TAOPP list（2）
- 15.stack/heap/queue（0）
- 16.TAOPP tree（2）
- 17.TAOPP c/c++（2）
- 18.TAOPP function（2）
- 19.TAOPP algorithms（8）
- 20.number operations（1）
- 21.Essays（6）
- 22.Big Data Processing（5）
- 23.Redis/MongoDB（0）
- 24.data structures（12）
- 25.Red-black tree（7）
- 26.Image Processing（3）
- 27.Architecture design（4）
- 28.Source analysis（3）
- 29.Recommend&Search（4）
- 30.Machine L & Deep Learning（19）

博客专栏

```
    //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）

BBC ABCDAB ABCDABCDABDE
ABCDABD

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）

BBC ABCDAB ABCDABCDABDE
ABCDABD

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）

BBC ABCDAB ABCDABCDABDE
ABCDABD

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

BBC ABCDAB ABCDABCDABDE
ABCDABD

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）

BBC ABCDAB ABCDABCDABDE
ABCDABD

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。



数据挖掘十大算法系列
文章：21篇
阅读：2555402



微软面试100题系列by July
文章：18篇
阅读：3237026



程序员编程艺术
文章：32篇
阅读：2472669



经典算法研究
文章：31篇
阅读：3345744

文章搜索

阅读排行

支持向量机通俗导论（理

(642765)

程序员面试、算法研究、

(546616)

教你如何迅速秒杀掉：9%

(511624)

从B树、B+树、B*树谈到

(460341)

十道海量数据处理面试题

(324038)

九月十月百度人搜，阿里

(287035)

从头到尾彻底理解KMP（

(285746)

十一、从头到尾解析Has

(265129)

教你初步了解红黑树

(260081)

横空出世，席卷互联网--i

(227235)

评论排行

支持向量机通俗导论（理

(582)

程序员面试、算法研究、

(515)

从头到尾彻底理解KMP（

(396)

九月十月百度人搜，阿里

(392)

从B树、B+树、B*树谈到

(353)

九月腾讯，创新工场，海

(351)

当今世界最为经典的十大

(327)

横空出世，席卷互联网--i

(292)

教你如何迅速秒杀掉：9%

(285)

教你初步了解红黑树

(270)

最新评论

从B树、B+树、B*树谈到R 树

暗黑-黑暗: 博主你好，按照文中删除B树节点的思路，我感觉如果一开始要删除的节点是M，就走不通啊；因为删除M后，两...

从B树、B+树、B*树谈到R 树

lengtianxiong: O（logn）是求得多少得次第，是2得，还是10得

算法面试：精选微软经典的算法

DisFney: 厉害

十、从头到尾彻底理解傅里叶变

BBC ABCDAB ABCDABCDABDE

ABCDABD

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，保过修改j 的位置，让模式串尽量地移动到有效的位置。

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3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)
    {
        //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
        {
            //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）。

http://blog.csdn.net/v_july_v/article/details/7041827

3/25

v_JULY_v: @qq_35189264:一往无前虎山行 拨开云雾见光明。恭喜恭喜，也欢迎多多推荐哦。
A A

十、从头到尾彻底理解傅里叶变
qq_35189264: 发现了好东西，感谢楼主的这篇文章，让刚学完数字信号处理的我醍醐灌顶

CNN笔记：通俗理解卷积神经网络
Jack床长: 我最近在写一系列的人工智能教程，通俗易懂，无需很高的数学基础，教程也力求风趣幽默，倡导快乐学习，欢迎...

从头到尾彻底理解KMP（2014年
Fizz彬彬: @v_JULY_v:必须的，July这花名还牢记

从头到尾彻底理解KMP（2014年
v_JULY_v: @ice_bin123:那是必须的，叫老师如何不欣慰。欢迎推荐给身边更多人，早日铲除他们心头困惑。^...

从头到尾彻底理解KMP（2014年
Fizz彬彬: 看《算法4》头大想起July的博客一看居然个把小时就摸清了，这大概就是当老师最大的欣慰吧

支持向量机通俗导论（理解SVM
努力学习的高高: 感谢您的整理与分享，非常受用！！谢谢您

我的驻点

00、我的新浪微博

01、我的Github主页

02、七月在线

03、寒

04、冯

05、zxy

06、廖雪峰

07、左耳朵耗子

07、zyx

08、阮一峰

09、唐巧的技术博客

10、sumnous

11、caopengcs

12、joycewyj的机器学习笔记

13、Harry

14、酷勤网

15、IT面试论坛

16、北大朋友的挖掘乐园

17、跟Rachel-Zhang一起读硕士

18、51nod

19、韩寒

20、曾经的叛逆与年少

21、code4app:iOS代码示例

22、Memory Model与并发编程

文章存档

2017年05月 (1)

2017年03月 (1)

2017年01月 (1)

2016年11月 (1)

2016年10月 (2)

展开

BBC ABCDAB ABCDABCDABDE

ABCDABD

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）。

BBC ABCDAB ABCDABCDABDE

ABCDABD

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

匹配失败，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
ABCD A	A,AB,ABC,ABCD	A,DA,CDA,BCDA	1
ABCDAB	A,AB,ABC,ABCD,ABCD A	B,AB,DAB,CDAB,BCDAB	2
ABCDABD	A,AB,ABC,ABCD,ABCD A 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 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：

BBC ABCDAB ABCDABCDABDE
ABCDABD

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：

BBC ABCDAB ABCDABCDABDE
ABCDABD

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。

BBC ABCDAB ABCDABCDABDE
ABCDABD

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。

BBC ABCDAB ABCDABCDABDE
ABCDABD

4. A与空格失配，向右移动1 位。

BBC ABCDAB ABCDABCDABD
ABCDABD

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。

BBC ABCDAB ABCDABCDABDE
ABCDABD

6. 经历第5步后，发现匹配成功，过程结束。

BBC ABCDAB ABCDABCDABDE
ABCDABD

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，两者得出出来的I 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

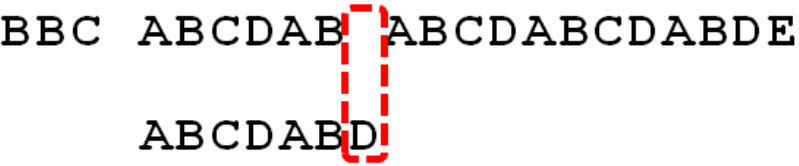
接下来，咱们来写代码求下next 数组。

基于之前的理解，可知计算next 数组的方法可以采用递推：

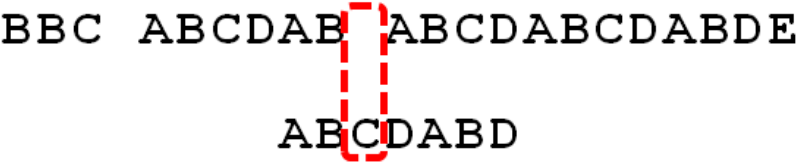
1. 如果对于值k，已有p0 p1, ..., pk-1 = pj-k pj-k+1, ..., pj-1，相当于next[j] = k。

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“ $p_{j-t} p_{j-t+1}, \dots, p_{j-1} p_j$ ”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 $next$ 数组（ $next[0, \dots, k, \dots, j]$ ）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“ $p_0 p_{k-1} = p_{j-k} p_{j-1} = AB$ ，可以看出 k 为2），现要求 $next[j+1]$ 等于多少？因为 $p_k = p_j = C$ ，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。

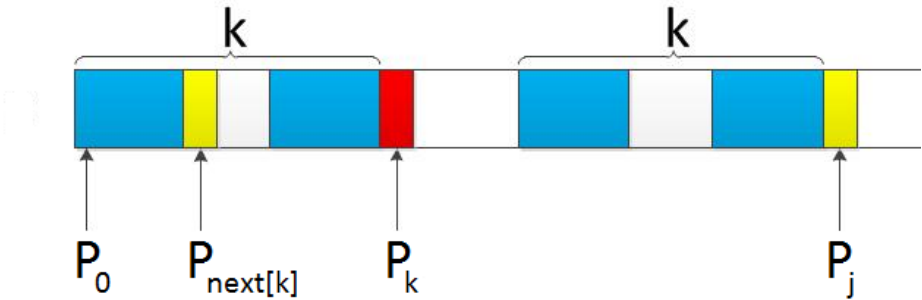
模式串	A	B	C	D	A	B	C	E
前后缀相同长度	0	0	0	0	1	2	3	0
next 值	-1	0	0	0	0	1	2	?
索引	p_0	p_{k-1}	p_k	p_{k+1}	p_{j-k}	p_{j-1}	p_j	p_{j+1}

但如果 $p_k \neq p_j$ 呢？说明“ $p_0 p_{k-1} p_k \neq p_{j-k} p_{j-1} p_j$ ”。换言之，当 $p_k \neq p_j$ 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“ $p_0 p_{k-1} p_k$ ”中不断的递归前缀索引 $k = next[k]$ ，找到一个字符 $p_{k'}$ 也为D，代表 $p_{k'} = p_j$ ，且满足 $p_0 p_{k'-1} p_{k'} = p_{j-k'} p_{j-1} p_j$ ，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 $p_0 p_{k-1} p_k$ 去跟后缀 $p_{j-k} p_{j-1} p_j$ 匹配，如果 p_k 跟 p_j 失配，下一步就是用 $p[next[k]]$ 去跟 p_j 继续匹配，如果 $p[next[k]]$ 跟 p_j 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 p_j 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	<u>D</u>	A	B	C	D	A	B	<u>D</u>	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，因为在000k处发现有个字符D， $p_0 = p_j$ ，所以pj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

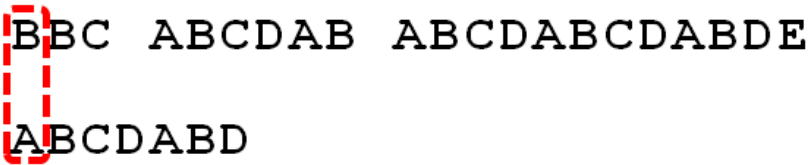
从上述表格可以看出，无论是之前通过“最长相同前缀后缀长度值右移一位，然后初值赋为-1”得到的next 数组，还是之后通过代码递推计算求得的next 数组，结果是完全一致的。

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 $j \neq -1$ ，且当前字符匹配失败（即 $S[i] \neq P[j]$ ），则令 i 不变， $j = \text{next}[j]$ 。此举意味着失配时，模式串P相对于文本串S向右移动了 $j - \text{next}[j]$ 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： $j - \text{next}[j]$** ，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 $j \neq -1$ ，且当前字符匹配失败（即 $S[i] \neq P[j]$ ），则令 i 不变， $j = \text{next}[j]$ ”，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 $i++$ ， $j = 0$ ，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 $i++$ ， $j++$ ”。

BBC ABCDAB ABCDABCDABDE
ABCDABD

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 $S[10] \neq P[6]$ ），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： $j - \text{next}[j] = 6 - 2 = 4$ 位。

BBC ABCDAB ABCDABCDABDE
ABCDABD

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： $j - \text{next}[j] = 2 - 0 = 2$ 位。

BBC ABCDAB ABCDABCDABDE
ABCDABD

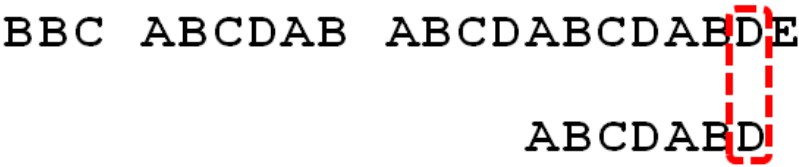
4. 移动两位之后，A 跟空格不匹配，模式串后移1 位。

BBC ABCDAB ABCDABCDABDE
ABCDABD

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 $j - \text{next}[j] = 6 - 2 = 4$ 位。

BBC ABCDAB ABCDABCDABDE
ABCDABD

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，等价于已匹配字的上一位字符所对应的最大长度值。原因是：

- 1. 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；
- 2. 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。

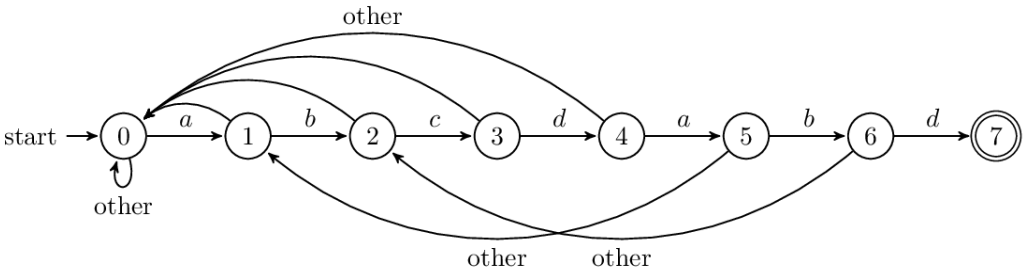
但为何本文不直接利用next 数组进行匹配呢？因为next 数组不好求，而一个字符串的前缀后缀的公共元素的最大长度值很容易求。例如若给定模式串“ababa”，要你快速口算出其next 数组，乍一看，每次求对应字符的next值时，还得把该字符排除之外，然后看该字符之前的字符串中有最大长度为多大的相同前缀后缀，此过程不够直接。而如果你求其前缀后缀公共元素的最大长度，则很容易直接得出结果：0 0 1 2 3，如下表格所示：

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

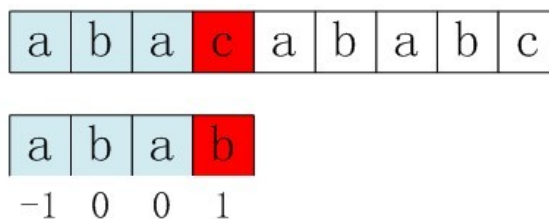
next 负责把模式串向前移动，且当第 j 位不匹配的时候，用第 $next[j]$ 位和主串匹配，就像打了张“表”。此外，next 也可以看作有限状态自动机的状态，在已经读了多少字符的情况下，失配后，前面读的若干个字符是有用的。



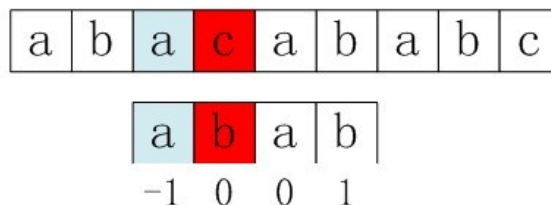
3.3.8 Next 数组的优化

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 $p[\text{next}[3]] = p[1] = b$ 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 $p[j] = p[\text{next}[j]]$ 。为什么呢？理由是：当 $p[j] \neq s[j]$ 时，下次匹配必然是 $p[\text{next}[j]]$ 跟 $s[j]$ 匹配，如果 $p[j] = p[\text{next}[j]]$ ，必然导致后一步匹配失败（因为 $p[j]$ 已经跟 $s[j]$ 失配，然后你还用跟 $p[j]$ 等值的 $p[\text{next}[j]]$ 去跟 $s[j]$ 匹配，很显然，必然失配），所以**不能允许 $p[j] = p[\text{next}[j]]$** 。如果出现了 $p[j] = p[\text{next}[j]]$ 咋办呢？如果出现了，则需要再次递归，即令 $\text{next}[j] = \text{next}[\text{next}[j]]$ 。

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p _j 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]]
优化的next数组	-1	0	-1	

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。

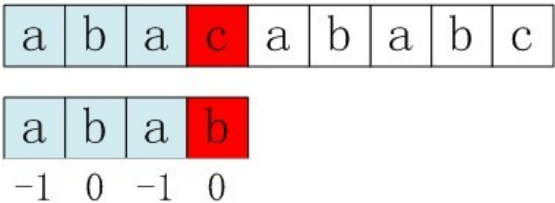
对于优化后的next数组可以发现一点：如果模式串的后缀跟前缀相同，那么它们的next值也是相同的，例如模式串abcabc，它的前缀后缀都是abc，其优化后的next数组为：-1 0 0 -1 0 0，前缀后缀abc的next值都为-1 0 0。

然后引用下之前3.1节的KMP代码：

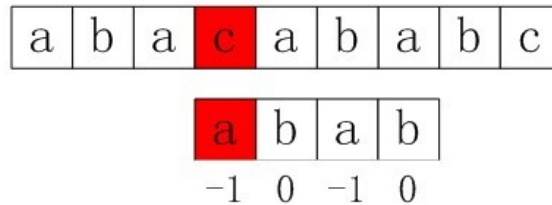
```
int KmpSearch(char* s, char* p)
{
    int i = 0;
    int j = 0;
    int sLen = strlen(s);
    int pLen = strlen(p);
    while (i < sLen && j < pLen)
    {
        //如果j = -1, 或者当前字符匹配成功（即S[i] == P[j]），都令i++, j++
        if (j == -1 || s[i] == p[j])
        {
            i++;
            j++;
   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

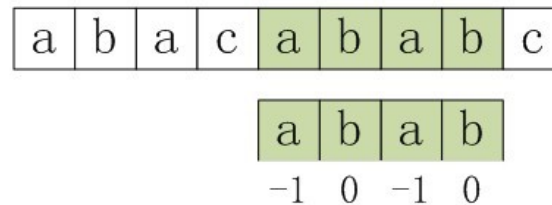
1. S[3]与P[3]匹配失败。



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 $i=3$, $j=\text{next}[0]=-1$, 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 $j=\text{pLen}$, 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。

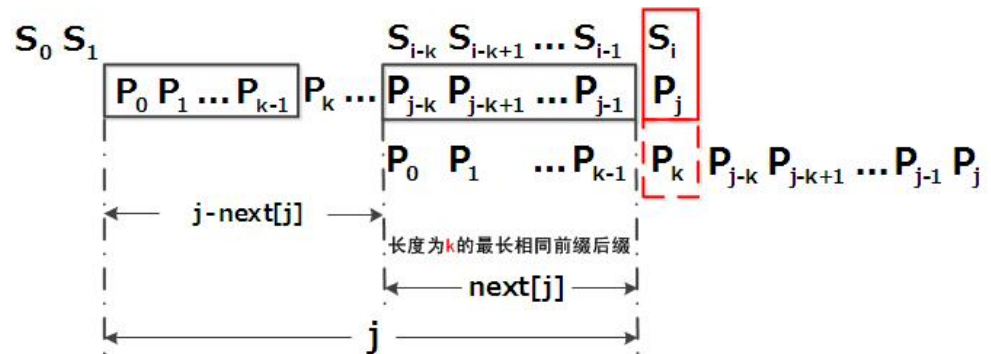


3.4 KMP的时间复杂度分析

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:

1. 如果模式串中存在相同前缀和后缀, 即 $p_{j-k} p_{j-k+1} \dots, p_{j-1} = p_0 p_1, \dots, p_{k-1}$, 那么在 p_j 跟 s_i 失配后, 让模式串的前缀 $p_0 p_1 \dots p_{k-1}$ 对应着文本串 $s_{i-k} s_{i-k+1} \dots s_{i-1}$, 而后让 p_k 跟 s_i 继续匹配。
2. 之前本应是 p_j 跟 s_i 匹配, 结果失配了, 失配后, 令 p_k 跟 s_i 匹配, 相当于 j 变成了 k , 模式串向右移动 $j - k$ 位。
3. 因为 k 的值是可变的, 所以我们用 $\text{next}[j]$ 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 , p_j 跟 s_i 失配时, 用 $p[\text{next}[j]]$ 继续跟 s_i 匹配, 相当于 j 变成了 $\text{next}[j]$, 所以, $j = \text{next}[j]$, 等价于把模式串向右移动 $j - \text{next}[j]$ 位。
4. 而 $\text{next}[j]$ 应该等于多少呢? $\text{next}[j]$ 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 $\text{next}[j] = k$ 。

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 $j \neq -1$, 且当前字符匹配失败 (即 $S[i] \neq P[j]$), 则令 i 不变, $j = \text{next}[j]$ 。此举意味着失配时, 模式串P相对于文本串S向右移动了 $j - \text{next}[j]$ 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 $i++$, $j++$; 如果匹配失败, i 不变 (即 i 不回溯), 模式串会跳过匹配过的 $\text{next}[j]$ 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. 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

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”。文本串找模式串是否存在于文本串中，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。

HERE IS A SIMPLE EXAMPLE
EXAMPLE

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。

HERE IS A SIMPLE EXAMPLE
EXAMPLE

HERE IS A SIMPLE EXAMPLE
EXAMPLE

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。

HERE IS A SIMPLE EXAMPLE
EXAMPLE

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？

HERE IS A SIMPLE EXAMPLE
EXAMPLE

HERE IS A SIMPLE EXAMPLE
EXAMPLE

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 - 好后缀在文本串中再次出现的位置，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。

HERE IS A SIMPLE EXAMPLE
EXAMPLE

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。因为是最末一位就失配，尚未获得好后缀。

HERE IS A SIMPLE EXAMPLE
EXAMPLE

由上可知，BM算法不仅效率高，而且构思巧妙，容易理解。

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：

substring searching algorithm

search

^

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高。完。

良

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%9393Morris%E2%80%9393Pratt_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://v.atob.site/kmp-next.html>。

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。

顶

324

踩

33

上一篇 Nginx源码剖析之内存池，与内存管理

下一篇 编程艺术第二十三~四章&十一续：杨氏矩阵查找，倒排索引关键词Hash编码

相关文章推荐

- Java实现KMP算法
 - 用户画像系统应用与技术解析--汪剑
 - KMP算法详解
 - 实时流计算平台Blink在阿里集团的应用实践--陈守元
 - KMP算法的简单总结以及java代码实现
 - Java 9新特性解读
 - 经典算法研究系列：五、红黑树算法的实现与剖析
 - Cocos2d-x 实战演练基础篇
- 教你如何迅速秒杀掉：99%的海量数据处理面试题
 - Unity3D移动端实战经验分享
 - 红黑树
 - 程序员如何转型AI工程师--蒋涛
 - 一步一图一代码，一定要让你真正彻底明白红黑树
 - 红黑二叉树详解及理论分析
 - 一步一图一代码，一定要让你真正彻底明白红黑...
 - 从头到尾彻底理解KMP（2014年8月22日版），

查看评论

70楼 [Fizz彬彬](#) 2017-09-19 20:09发表



看《算法4》头大想起July的博客一看居然个把小时就摸清了，这大概就是当老师最大的欣慰吧

Re: [v_JULY_v](#) 2017-09-19 22:56发表



回复ice_bin123：那是必须的，叫老师如何不欣慰。欢迎推荐给身边更多人，早日铲除他们心头困惑。^__^

Re: [Fizz彬彬](#) 2017-09-20 09:40发表



回复v_JULY_v：必须的，July这花名还好记

69楼 [执念墨尘枫](#) 2017-09-06 17:09发表



已经看第二次了 之前学过一次 现在看印象更深刻，建议看两遍

68楼 [执念墨尘枫](#) 2017-09-06 17:07发表



<http://blog.csdn.net/column/details/17206.html> 玩ACM的可以看看

67楼 [gainsuper](#) 2017-08-22 16:54发表



把作者里面KMP算法的运行流程写了一下，相信可以帮助更多的人理解。

66楼 [gainsuper](#) 2017-08-22 16:53发表



i	j	s[i]	p[j]	next[j]	j = -1 s[i] == p[j]	else	i < sLen && p < pLen
0	0	B	A	-1		j = next[j]	
0	-1	B	\N	\N	i++;j++;		
1	0	B	A	-1		j = next[j]	
1	-1	B	\N	\N	i++;j++;		
2	0	C	A	-1		j = next[j]	
2	-1	C	\N	\N	i++;j++;		
3	0	\s	A	A		j = next[j]	
3	-1	\s	\N	\N	i++;j++;		
4	0	D	A	-1		j = next[j]	
4	-1	D	\N	\N	i++;j++;		
5	0	A	A	-1	i++;j++;		
6	1	B	B	0	i++;j++;		
7	2	C	C	0	i++;j++;		
8	3	D	D	0	i++;j++;		
9	4	A	A	0	i++;j++;		
10	5	B	B	1	i++;j++;		
11	6	\s	D	2		j = next[j]	
11	2	\s	C	0		j = next[j]	
11	0	\s	D	-1		j = next[j]	
11	-1	\s	\N	\N	i++;j++;		
12	0	A	A	-1	i++;j++;		
13	1	B	B	0	i++;j++;		
14	2	C	C	0	i++;j++;		
15	3	D	D	0	i++;j++;		
16	4	A	A	0	i++;j++;		
17	5	B	B	1	i++;j++;		
18	6	C	D	2		j = next[j]	
18	2	C	C	0	i++;j++;		
19	3	D	D	0	i++;j++;		
20	4	A	A	0	i++;j++;		
21	5	B	B	1	i++;j++;		
22	6	D	D	2	i++;j++;		
23	7	E	\N	\N			break

65楼 [无影风Victor](#) 2017-08-19 17:43发表



看了许多博客，终于理解。多谢

Re: [v_JULY_v](#) 2017-08-19 18:25发表



回复victorzzzz：恭喜恭喜，我之前也有同样的豁然开朗的感觉，也欢迎推荐给身边更多人，^__^

64楼 [海棠_依旧](#) 2017-08-18 01:07发表



BM算法老师没有实现,我用Java实现啦,大家有兴趣的话看一看一下,同时也非常感谢老师的讲解,讲的真的很棒很清楚!

亲测思路+实现正确!

<https://github.com/pingcai/Algorithm/blob/master/src/algorithm/string/BM.java>

Re: [v_JULY_v](#) 2017-08-18 22:12发表



回复huangpingcai：没细看代码，但很赞实践动手力！~

63楼 [Twiliz](#) 2017-08-14 16:56发表



博主的讲解十分详细，受益匪浅

62楼 [purple329](#) 2017-08-08 11:52发表



首先，博主写得是非常好的，但我在如下段落中有个疑问：

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， $next[j + 1] = 0$ 。

这里的 k' 描述好像有点问题吧？从文字中看， k' 代表的是下标，也是 $next[k]$ 的值(假定第1次递归前缀索引 $k' = next[k]$)，所以 $next[j + 1] = k' + 1 = next[k'] + 1$ 应改为 $next[j + 1] = k' + 1 = next[k] + 1$ ，是这样吧？

原来写的是 $k' + 1 = next[k'] + 1$ ，即 $k' = next[k]$ ，从DABCDABDE这个串来看， k' 为0， $next[k]$ 为-1，显然不等。

之前看了好几次这个博文，对 $k = next[k]$ 一直是不太理解，因为 $next$ 数组给人的感觉就是最大相同长度的前缀后缀，并没有和数组下标有什么关联，但其实两者是有着深刻联系的。

希望这一点能够再深入挑明一些。谢谢博主的奉献

61楼 [jingzhongchen](#) 2017-07-28 21:38发表



看到博主的算法功底如此深厚，而算法却是我薄弱的环节，导致在我制作出函数链后，算法函数一直还没写。

看到博主算法这么好，想邀请你去函数链写算法函数。

这样也可以给你带来一些便利：在你写算法博客的时候，可以在函数链提供在线演示运行的例子，加深读者对算法的理解。具体情况请看：<https://hanshulian.com/nodes?type=recent>

60楼 [fanmula](#) 2017-07-13 14:39发表



写得很棒！

59楼 [Felix吴](#) 2017-06-19 13:41发表



谢谢博主，受益匪浅。

读罢只有一个感觉：KMP毕业了。

本质简单的东西要讲清楚却不是一件容易的事情，在学习数据结构的时候这种感觉尤其明显。需要静下心，慢慢的就柳暗花明了。

Re: [v_JULY_v](#) 2017-06-19 14:24发表



回复redocx：是的，不可急功近利哦，一时的“偷工减料”，日后需要花多倍时间去弥补。搞学问，贵在踏踏实实、一步一个脚印。

58楼 [Felix吴](#) 2017-06-19 13:40发表



谢谢博主，受益匪浅。

读罢只有一个感觉：KMP毕业了。

本质简单的东西要讲清楚却不是一件容易的事情，在学习数据结构的时候这种感觉尤其明显。需要静下心，慢慢的就柳暗花明了。

57楼 [LaoJiu_](#) 2017-06-04 12:12发表



写的很好，但是排版不好，可以看下我的这篇<http://www.61mon.com/index.php/archives/183/>

56楼 [rudy_yuan](#) 2017-05-13 21:40发表



博主您好。您文中提到的参考文献17，原文地址已经修改为<http://v.atob.site/kmp-next.html> 我之前的那个域名已经过期了，麻烦您有空修改下，不好意思，谢谢。

我就是这篇文章的作者。这个新域名我会长期管理的，多谢啦。



20/25

43楼 [siyanyu2011](#) 2016-08-28 14:32发表



写的很棒，如何文中能对求next时候，为什么 $k = \text{next}[k]$ 做更相信明白的分析就更好了，自己看了好久才明白因为对称原理，才直接另 $k = \text{next}[k]$ ，让 $p[k]$ 这个前缀做比较。

42楼 [红炎背理](#) 2016-08-03 19:42发表



第一次看了一半放弃了，今天终于看完了KMP算法，收获颇深，谢谢博主！！

Re: [v_JULY_v](#) 2016-08-03 23:05发表



回复jojojoan：恭喜恭喜。坚持一下，必将受益匪浅。欢迎分享给身边更多人

41楼 [Switch_vov](#) 2016-07-07 13:20发表



赐教了

40楼 [记忆力不好](#) 2016-06-20 13:06发表



写的太长了

39楼 [菜菜粥](#) 2016-06-05 14:25发表



算法就是牛逼，慢慢爬坡

38楼 [shang_1991](#) 2016-05-09 13:43发表



感觉还是没讲清楚

37楼 [失明后的世界](#) 2016-05-07 14:12发表



看懂了递归前缀索引 $k = \text{next}[k]$ 这一步 有种瞬间明白的感觉！！看其他博客看的我一头雾水 终于明白 好想哭！！感谢！

36楼 [AirDDD](#) 2016-05-03 17:29发表



写的太好辣

35楼 [iido1](#) 2016-04-17 23:42发表



很感谢楼主的精心一步步的分析，终于明白了KMP算法的精髓， $\text{next}[]$ 数组推算过程、以及优化后的 $\text{next}[]$ 数组，花了一个小时来拜读，很感谢！

34楼 [量子使徒](#) 2016-04-02 03:24发表



写得真不错，希望楼主的书能大卖

33楼 [少有人走的路上](#) 2016-01-25 17:01发表



博主讲得确实够详细，但是3.2和3.3确实内容太重复了。你把 next 数组讲得很透，但内容实在是太冗长了， next 数组被你复杂化了。重要的是，你一来3.2的模式串里abab的第二个a的前后缀公共长度就求错了，误导性很强。

Re: [少有人走的路上](#) 2016-01-25 17:10发表



回复androidchanhao：不好意思，那个模式串前后缀公共长度是没有错的，但文章确实过于冗长了，这个观点不变。

32楼 [SUSTCer_章鱼烧](#) 2016-01-16 15:58发表



今天终于恍然大悟优化版的 next 数组就是为了避免计算后下表对应的字符和失配的字符是一样的。

Re: [SUSTCer_章鱼烧](#) 2016-01-16 16:10发表



回复kids412kelly：不过还是不太理解为什么计算 next 数组时，需要用 $k = \text{next}[k]$ 来控制 k 的变化？博主能解释一下吗？谢谢。

Re: [SUSTCer_章鱼烧](#) 2016-01-16 16:29发表



回复kids412kelly：彻底理解了。。。因为不满足 if 的条件说明前缀中与和后缀失配的字符对应的 $p[\text{next}[i]]$ 和前缀中的失配字符依然不同，还是不能匹配，所以相当于继续递归去寻找不同的字符了。博主好棒！

31楼 [ymymrydzh1](#) 2016-01-07 23:01发表



KMP后边跟着的这个优化：问题出在不该出现 $p[j] = p[\text{next}[j]]$ 。有什么意义呢？就是加了一次判断来处理这个必然不等的情况，如果不加，无非就是把把这个判断放到下一次循环里去做了。所以就算当前字符不等，只是让当前这次移动模式串更加精确，但并没有减少判断次数，没有看到哪里优化了。而且还让所有其他情况多了一次判断。

Re: [buotctt](#) 2016-04-11 15:50发表



回复ymymrydz1：在模式串上的操作是一次性的
对于匹配串可能很长，在匹配时出现很多次 $p[j]=p[next[k]]$ 情况。
优化后就不会出现。

30楼 q1097306512 2015-12-11 10:22发表



早看这篇博客的话，就不用费那么多功夫了，感谢July，不知道编程之法上会不会因为篇幅有限讲解得这么仔细

Re: v_JULY_v 2015-12-11 10:47发表



回复q1097306512：你好，为确保通俗易懂，《编程之法》第4.4节 KMP讲得很详细，且新书质量高于博客。
欢迎推荐分享给身边更多人、让更多人受益。(*^__^*)

29楼 zhihua_bupt 2015-12-03 20:38发表



讲的很详细，谢谢july

Re: v_JULY_v 2015-12-04 12:07发表



回复geekmanong：不客气，更好更优化的KMP算法版本收录于新书《编程之法：面试和算法心得》
京东有售：<http://item.jd.com/11786791.html>

28楼 xiinsist 2015-10-29 10:42发表



great

27楼 zhaoyunfullmetal 2015-10-27 16:06发表



楼主威武啊，恭喜新书顺利初版。 我的疑问是，一个字符串的next数组大小是否应该比字符串的长度大1，比如说模式串：
aaa,它的未优化的next数组为： $\{-1,0,1\}$ 。 但是在源串 $s="aaabaaaac"$ 比较模式串出现的次数（多次比较）， $\{-1,0,1\}$ 的next数组
是不够用的，因此我们要求得比模式字符串大1的next数组： $\{-1,0,1,2\}$ ，这样在第一次匹配成功之后模式串index j才能继续跳
转到 $next[3]=2$ 的位置。
大神怎么看撒？

26楼 麻木了 2015-10-10 15:34发表



博主写的好难懂

25楼 qq_24683561 2015-10-10 14:05发表



暴力匹配算法：第六小点有误
我觉得用这句话（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
 $P[1] \neq P[0]$ ，故S[5]必定不等于P[0]）
引出（所以回溯过去必然会导致失配）这个地方有误，如果你举得例子当中 $S[4]='A'$ ， $S[5]='A'$
 $P[0]='A'$ ， $P[1]='A'$ ，那么当你后面发生不匹配的时候，i进行回溯后，S[5]跟P[0]是匹配的
所以这句话我觉得不妥

24楼 qq_24683561 2015-10-10 14:04发表



暴力匹配算法：第六小点有误
我觉得用这句话（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
 $P[1] \neq P[0]$ ，故S[5]必定不等于P[0]）
引出（所以回溯过去必然会导致失配）这个地方有误，如果你举得例子当中 $S[4]='A'$ ， $S[5]='A'$
 $P[0]='A'$ ， $P[1]='A'$ ，那么当你后面发生不匹配的时候，i进行回溯后，S[5]跟P[0]是匹配的
所以这句话我觉得不妥

23楼 biill 2015-08-28 12:53发表



如果要连续匹配下一个模式的话，数组next是不能优化的。循环 $k = next[k]$ 应当在KMP匹配算法里进行。

22楼 六爻 2015-08-17 11:50发表



博主，你好，十分感谢你写的博客，让我对KMP算法的理解更加深刻，不过好像有一个地方写错了。在文章3.1那一部分的代码
段后面的哪一段文字第二行——“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 $j \neq -1$ ，且当前字符匹配失败（即 $S[i] \neq P[j]$ ），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 $j - next[j]$ （ $j - next[j] = 6 - 2 = 4$ ）。”
当当前字符匹配失败时的后面应该是，i不变， $j = next[j-1]$ 才对，对于为优化之前的KMP算法，用你上面的例子中的子
串“ABCDABD”，它的next数组值应该分别为 $[0,0,0,0,1,2,0]$ 。所以多余当前未匹配的字符，j应该取它上一个匹配的字符 $next[j-1]$ 的
值，也就是向后移动， $j = next[j-1]$ 位才对。

Re: v_JULY_v 2015-08-17 17:38发表



回复YZS_L_H：你好，原文没错。
“ABCDABD”，它的next数组值应该分别为 $[-1,0,0,0,1,2]$ 。
至于你说的 $[0,0,0,0,1,2,0]$ 非标准的next数组，而是最大前缀后缀公共元素长度。

更进一步，你可以再看下全文，或体会下3.3.3节。

①当根据最大长度 $[0,0,0,0,1,2,0]$ 来匹配时，

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

②当根据next数组 [-1,0,0,0,1,2] 进行匹配时，

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。

两相比较，①中的 $j - \text{next}[j-1]$ 完全等同于②中的 $j - \text{next}[j]$ 。

换言之： j 从0计数，所以已匹配的字符数 = 失配字符所在位置，且①中的 $\text{next}[j-1]$ 等同于②中的 $\text{next}[j]$ 。

而我原文3.1节那里的描述是根据② next数组进行匹配。

综上，原文3.1节没问题。

21楼 [NK_test](#) 2015-08-16 10:55发表



终于理解了KMP，谢谢楼主

20楼 [lirui9601](#) 2015-08-14 21:01发表



你好，July，我想3.3.1下面的配图是否用真前缀和真后缀代替“前缀”，“后缀”比较好

19楼 [从来不作](#) 2015-08-11 19:25发表



“那为何递归前缀索引 $k = \text{next}[k]$ ，就能找到长度更小的相同前缀后缀呢？这又归根到next数组的含义。”我还是没看懂next数组的求法。要哭了。。。

Re: [v_JULY_v](#) 2015-08-12 17:50发表



回复banshichiqinglangzi：你好，建议你好好结合这段话：

“我们拿前缀 $p_0 p_{k-1} p_k$ 去跟后缀 $p_{j-k} p_{j-1} p_j$ 匹配，如果 p_k 跟 p_j 失配，下一步就是用 $p[\text{next}[k]]$ 去跟 p_j 继续匹配，如果 $p[\text{next}[k]]$ 跟 p_j 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 $p[\text{next}[\text{next}[k]]]$ 去跟 p_j 匹配。此过程相当于模式串的自我匹配，所以不断的递归 $k = \text{next}[k]$ ，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。”

和下文的例子（DABCDABDE）、图表理解。一点点来。

Re: [rudy_yuan](#) 2015-08-11 21:55发表



回复banshichiqinglangzi：关于next数组的求法，你可以看看我的这篇博客，一个图说明一切。。仔细体会下就会恍然大悟。

<http://www.rudy-yuan.net/archives/182/>

18楼 [Pekary](#) 2015-07-27 10:34发表



希望博主能够讲一讲扩展KMP算法，期待您的回复。

17楼 [奔跑的小河](#) 2015-07-25 20:44发表



写的太好了，KMP不是一种很好的字符串模式匹配算法，但是它提供了一个很好的匹配思路，所以我觉得没有必要总结的太长，我自己也总结了一篇比较精简的，自我感觉良好，<http://blog.csdn.net/z702143700/article/details/46945509>。可以看看。

16楼 [go_og](#) 2015-07-23 10:23发表



剖析的入木三分，读一遍就完全明白~万分感谢！

15楼 [abcd1f2](#) 2015-07-22 18:35发表



先凑凑人气，下次在仔细看

14楼 [To-Big_Fish](#) 2015-07-05 11:37发表



写的真好！！！！

不知道实际应用中BM算法和KMP算法那一种多一些~

13楼 [张美红](#) 2015-06-28 10:42发表



太棒了！

12楼 [Royecode](#) 2015-06-25 19:33发表



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 p_0 处发现也有个字符D， $p_0 = p_j$ ，所以 $p[j]$ 对应的长度值为1，相当于E对应的next 值为1。

博主，我怎么感觉这句话老理解不了，为什么说 $p_0 = p_j$ ，所以 $p[j]$ 对应的长度值是1？请赐教

Re: [v_JULY_v](#) 2015-06-25 20:06发表



回复Royecode：你好，换句话说，其实就是字符E之前的字符串“DABCDABD”中有长度为1的相同前缀和后缀：D。

11楼 [a790398207](#) 2015-05-29 22:25发表



3.3.4 通过代码递推算next 数组

中的第二个表

里面的字符E的next值不是2么？怎么会是0？

Re: v_JULY_v 2015-05-29 22:39发表



回复a790398207：你好，3.3.4 通过代码递推算next 数组中的第二个表里面的字符D的next值是2，字符E的next值为0。

PS：第二个表是为了说明 $pk \neq pj$ 的情况，所以令模式串为ABCDABDE。

具体计算过程在原文中，以及我在上一楼给你的回复中说的很清楚了。

Re: a790398207 2015-05-30 14:45发表



回复v_JULY_v：感谢楼主详细的解答^_^

10楼 a790398207 2015-05-29 21:22发表



还是不懂

原文：

假定给定模式串ABCDABCE，且已知 $next[j] = k$ （相当于“ $p_0 p_{k-1} = p_{j-k} p_{j-1} = AB$ ”，可以看出 k 为2）

如果此时 $pk \neq pj$ ，那么 $j+1$ 之前已经匹配到的字符不就是已知的2吗？为什么还要递归？

Re: v_JULY_v 2015-05-29 22:29发表



回复a790398207：你好，其实这非常好理解的。

假定给定模式串ABCDABCE，且已知 $next[j] = k$ （相当于“ $p_0 p_{k-1} = p_{j-k} p_{j-1} = AB$ ”，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。

上面这 $pk = pj$ 的情况是没问题的，对吧？

接下来看 $pk \neq pj$ 的情况。

如果 $pk \neq pj$ ，说明模式串可能就变成是：ABCDABDE

此时，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，咱们只能去递归寻找长度更短一点的相同前缀后缀。

进一步，虽然 $p_j = D$ 之前的模式串ABCDAB有长度为2的相同前缀后缀“AB”（即 p_j 的next值为2），但 $p_{j+1} = E$ 前的模式串ABCDABD没有长度为2的相同前缀后缀“AB”。

还有疑问？请仔细看下，ABCDABD的

前缀有：A、AB、ABC、ABCD、ABCDA、ABCDAB，

后缀有：D、BD、ABD、DABD、CDABD、BCDABD，

即压根就没有相同的前缀后缀。

9楼 JohnCusack 2015-05-27 11:34发表



引用“lzc5353389”的评论：

回复v_JULY_v：是我多虑了。本来想的是当 $next[j]$ 跳到 $next[...]$

我也有这方面的考虑，但还是不确定当 $next[j]$ 跳到 $next[k]$ 后，会不会再次出现 $p[j] == p[next[j]]$ 的情况。不知道您是怎么考虑的。

8楼 sesesessse 2015-05-11 23:31发表



楼组求next的时候给得代码是有误的。

当前缀 $k > j/2 + 1$ 的时候，会造成next求值错误。

比如求ababaa的数组，用楼主给得代码结果为-1,0,0,1,2,3；

目测的话， $next[5]$ 的结果为1，而代码求出的值给3

Re: 春秋非我 2015-05-14 11:26发表



回复u012148999：博主的算法是没有问题的，对于ababaa的数组， $next[5]$ 的结果的确是3，最长前后缀aba。

7楼 sesesessse 2015-05-11 23:29发表



楼组求next的时候给得代码是有误的。

当前缀 $k > j/2 + 1$ 的时候，会造成next求值错误。

比如求ababaa的数组，用楼主给得代码结果为-1,0,0,1,2,3；

目测的话， $next[5]$ 的结果为1，而代码求出的值给3

6楼 sesesessse 2015-05-11 23:28发表




楼组求next的时候给得代码是有误的。

当前缀 $k > j/2 + 1$ 的时候，会造成next求值错误。

比如求ababaa的数组，用楼主给得代码结果为-1,0,0,1,2,3；

目测的话， $next[5]$ 的结果为1，而代码求出的值给3

5楼 [kpy771224202](#) 2015-05-06 11:06发表




博主你好：

我想问一下暴力匹配中：


2、如果失配（即 $S[i] \neq P[j]$ ），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。 i 回溯为什么不是 $i++$ 。我自己测试 $i++$ 也是可以完成匹配的。一直无法理解 $i = i - (j - 1)$ 的推导过程，望指教，十分谢谢！

Re: [着数](#) 2015-05-17 15:30发表



回复kpy771224202：你理解错了，暴力匹配指遇到不匹配时， i 回溯到下一位，这里的下一位指的是匹配串的第二位（若从0开始数，即第1位）。也就是说， $j \neq 0$ 时，就不能 $++$ 了。


4楼 [sinuos](#) 2015-04-27 16:25发表



博主你好，对于next数组还是有点不太理解，文中的解释是“为了寻找长度相同的前缀后缀，我们拿前缀 $p_0 p_{k-1} p_k$ 去跟后缀 $p_{j-k} p_{j-1} p_j$ 匹配，如果 p_k 跟 p_j 失配，下一步就是用 $p_{next[k]}$ 去跟 p_j 继续匹配”，为什么 p_k 和 p_j 失配后，下一步就


跟 p_j 继续匹配

Re: [着数](#) 2015-05-17 15:37发表



回复takingfire09：因为你必须满足 $p_0 p_1 p_2 \dots p_{m-1}$ 与 $p_{j-mpj-m+1} \dots p_{j-1}$ 匹配啊，所以你必须去匹配 $p_{next[k]}$ 之间的值都无效，因为你无法满足上面的匹配要求。

3楼 [雪寒2216](#) 2015-04-20 11:06发表




讲的很细心，很明白，谢谢，之前也是看很多资料糊里糊涂的，偶尔明白了，一段时间也忘记了，楼主说在北京开设了算法学习班，不知道可否了解下

2楼 [liuhmmjj](#) 2015-03-31 10:59发表



真是太给力了

1楼 [liuhmmjj](#) 2015-03-31 10:58发表



谢谢大神分享啊

查看更多评论

发表评论

用户名：

a537656

评论内容：



提交

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

公司简介 | 招贤纳士 | 广告服务 | 联系方式 | 版权声明 | 法律顾问 | 问题报告 | 合作伙伴 | 论坛反馈