这篇长文历时近两天终于完成了，前些天帮伯乐在线翻译一篇文章《[为什么GNU grep如此之快？](http://blog.jobbole.com/52313/)》，里面提及到grep速度快的一个重要原因是使用了Boyer-Moore算法作为字符串搜索算法，兴趣之下就想了解这个算法，发现这个算法一开始还挺难理解的，也许是我理解能力不是很好吧，花了小半天才看懂，看懂了过后就想分享下，因为觉得这个算法真的挺不错的，以前一直以为字符串搜索算法中KMP算很不错的了，没想到还有更好的，Boyer-Moore算法平均要比KMP快3-5倍。

下面是我对该算法的理解，参考了一些关于该算法的介绍，里面每一张图都画的很认真，希望能讲清楚问题，有什么错误、疑问或不懂的地方麻烦大家一定要提出来，共同学习进步！下面正文开始。

## 简单介绍

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，具体推荐看下我最近的一篇译文“为什么GNU grep如此之快？”作者是GNU grep的编写者Mike Haertel。

## 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，是字符集大小，一般为256；
* 搜索阶段时间复杂度是；
* 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，比如在文本串bn中搜索模式串am-1b ，只需要n/m次比较。

**这些特征先让大家对该算法有个基本的了解，等看懂了算法再来看这些特征又会有些额外的收获。**

## 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：

|  |
| --- |
| **While (j <= strlen(text) - strlen(pattern)) {**  **for (i = 0; i < strlen(pattern) && pattern[i] == text[i + j]; ++i);**  **if (i == strlen(pattern)) {**  **Match;**  **break;**  **}**  **else**  **++j；**  **}** |

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：

|  |
| --- |
| **While (j <= strlen(text) - strlen(pattern)) {**  **for (i = strlen(pattern); i >= 0 && pattern[i] == text[i + j]; --i);**  **if (i < 0)) {**  **Match;**  **break;**  **}**  **else**  **j += BM()；**  **}** |

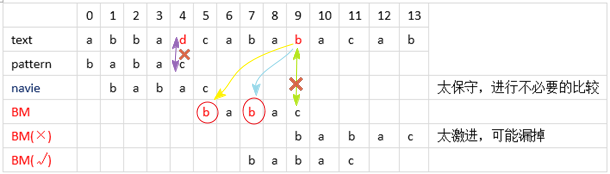
BM算法的精华就在于**BM(text, pattern)**,也就是BM算法当不匹配的时候一次性可以跳过不止一个字符。即它不需要对被搜索的字符串中的字符进行逐一比较，而会跳过其中某些部分。通常搜索关键字越长，算法速度越快。它的效率来自于这样的事实：**对于每一次失败的匹配尝试，算法都能够使用这些信息来排除尽可能多的无法匹配的位置。**即它充分利用待搜索字符串的一些特征，加快了搜索的步骤。

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）。

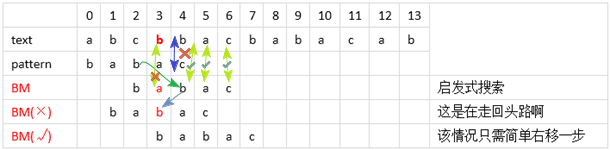
下面不直接书面解释这两个算法，为了更加通俗易懂，先用实例说明吧，这是最容易接受的方式。

## 字符串搜索头脑风暴

大家来头脑风暴下：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。

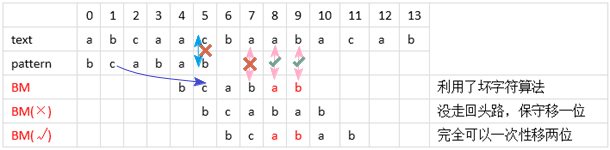


But, 如果遇到下面这样的情况，开始pattern中的c和text中的b不匹配，Ok，按上面的规则将pattern右移直至最右边的b与text的b对齐进行比对。再将pattern中的c与text中的c进行比对，匹配继续往左比对，直到位置3处pattern中的a与text中的b不匹配了，按上面讲的启发式规则应该将pattern中最右边的b与text的b对齐，可这时发现啥了？pattern走了回头路，干吗？当然不干，才不要那么傻，针对这种情况，只需要将pattern简单的右移一步即可，坚持不走回头路！

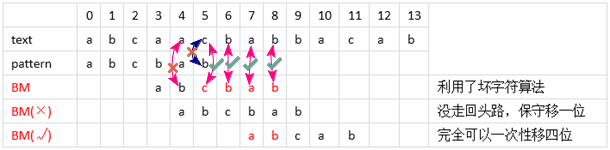


好了，这就是所谓的“坏字符算法”，简单吧，通俗易懂吧，上面用红色粗体字标注出来的b就是“坏字符”，即不匹配的字符，坏字符是针对text的。

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  :**mahtavaataloma**isema omalomailuun

模式串:**maisemaomaloma**

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。

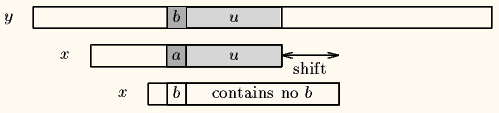
光用例子说明当然不够，太浅了，而且还不一定能完全覆盖所有情况，不精确。下面就开始真正的理论探讨了。

## BM算法理论探讨

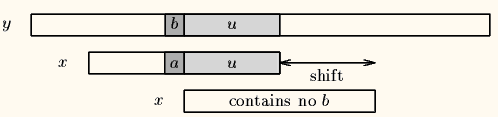
**（1）坏字符算法**

当出现一个坏字符b时, BM算法向右移动模式串,让模式串中最靠右的字符b与坏字符b对齐，然后重新开始匹配。坏字符算法有两种情况。

Case1：假定模式串中有对应的坏字符b，则让模式串中最靠右的字符b与文本串中的坏字符b对齐,当然,BM算法不可能走回头路，因为若是回头路，则移动距离就是负数了，考虑到移动距离的选取还有一个好字符规则,并且好字符规则各种情况下的移动距离都会大于0,所以这种情况下, 这种负的移动距离肯定不会被选择.

[](http://jbcdn2.b0.upaiyun.com/2013/12/BM-math05.png)

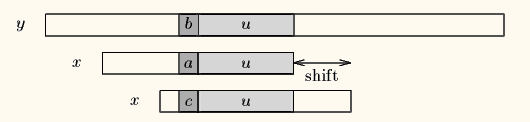
Case2：模式串中不存在坏字符，很好，**将模式串移动到文本串坏字符之后**，如下图。

[](http://jbcdn2.b0.upaiyun.com/2013/12/BM-math06.png)

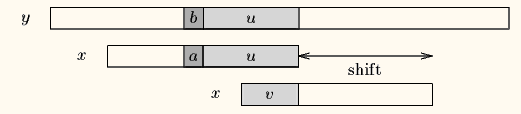
**（2）好后缀算法**

如果程序匹配了一个好后缀, 并且在模式串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，我需要移动才能匹配。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。

[](http://jbcdn2.b0.upaiyun.com/2013/12/BM-math07.png)

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]。

[](http://jbcdn2.b0.upaiyun.com/2013/12/BM-math08.png)

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算法是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。

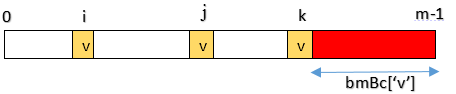
shift（好后缀）和shift（坏字符）通过模式串的预处理数组的简单计算得到。坏字符算法的预处理数组是bmBc[]，好后缀算法的预处理数组是bmGs[]。

## 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。

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，但事实上这样是不对的，因为i位置处的字符可能在pattern中多处出现（如下图所示），而我们需要的仅仅是**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）。

[](http://jbcdn2.b0.upaiyun.com/2013/12/BM-math09.png)

如前所述，bmBc[]的计算分两种情况，与前一一对应。

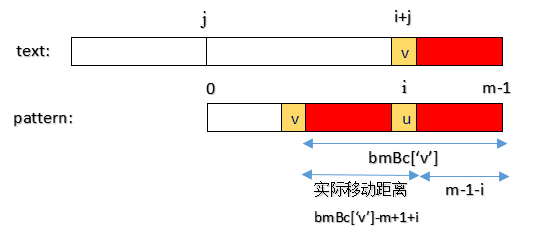
Case1：字符在模式串中有出现，**bmBc[‘v’]表示字符v在模式串中最后一次出现的位置，距离模式串串尾的长度**，如上图所示。

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：

|  |
| --- |
| **void PreBmBc(char \*pattern, int m, int bmBc[])**  **{**  **int i;**  **for(i = 0; i < 256; i++)**  **{**  **bmBc[i] = m;**  **}**  **for(i = 0; i < m - 1; i++)**  **{**  **bmBc[pattern[i]] = m - 1 - i;**  **}**  **}** |

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**。

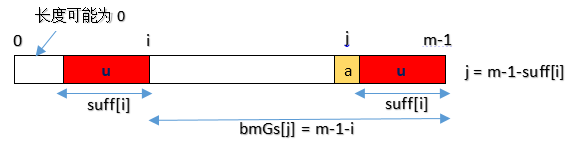
[](http://jbcdn2.b0.upaiyun.com/2013/12/BM-math10.png)

**（2）计算好后缀数组bmGs[]**

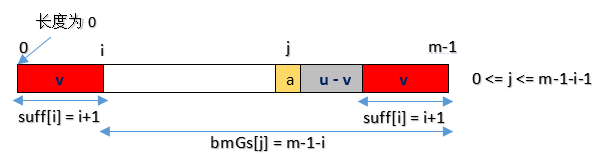
这里bmGs[]的下标是数字而不是字符了，表示字符在pattern中位置。

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。

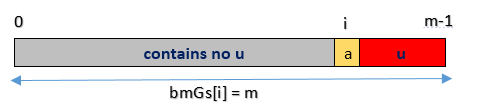
Case1：对应好后缀算法case1，如下图，j是好后缀之前的那个位置。

[](http://jbcdn2.b0.upaiyun.com/2013/12/BM-math11.png)

Case2：对应好后缀算法case2：如下图所示：

[](http://jbcdn2.b0.upaiyun.com/2013/12/BM-math13.png)

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

[](http://jbcdn2.b0.upaiyun.com/2013/12/BM-math14.png)

这样就更加清晰了，代码编写也比较简单：

|  |
| --- |
| **void PreBmGs(char \*pattern, int m, int bmGs[])**  **{**  **int i, j;**  **int suff[SIZE];**  **// 计算后缀数组**  **suffix(pattern, m, suff);**  **// 先全部赋值为m，包含Case3**  **for(i = 0; i < m; i++)**  **{**  **bmGs[i] = m;**  **}**  **// Case2**  **j = 0;**  **for(i = m - 1; i >= 0; i--)**  **{**  **if(suff[i] == i + 1)**  **{**  **for(; j < m - 1 - i; j++)**  **{**  **if(bmGs[j] == m)**  **bmGs[j] = m - 1 - i;**  **}**  **}**  **}**  **// Case1**  **for(i = 0; i <= m - 2; i++)**  **{**  **bmGs[m - 1 - suff[i]] = m - 1 - i;**  **}**  **}** |

So easy? 结束了吗？还差一步呢，这里的suff[]咋求呢？

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bcabab，以最后一个字符b为后缀的后缀串为bcababab，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bcababab，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：

|  |
| --- |
| **void suffix(char \*pattern, int m, int suff[])**  **{**  **int i, j;**  **int k;**  **suff[m - 1] = m;**  **for(i = m - 2; i >= 0; i--)**  **{**  **j = i;**  **while(j >= 0 && pattern[j] == pattern[m - 1 - i + j])**  **j--;**  **suff[i] = i - j;**  **}**  **}** |

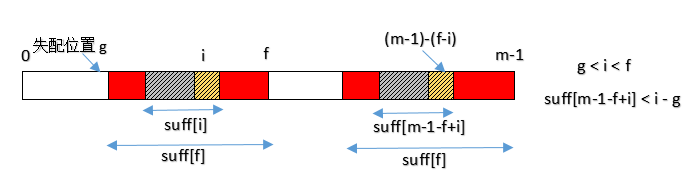
这样可能就万事大吉了，可是总有人对这个算法不满意，感觉太暴力了，于是有聪明人想出一种方法，对上述常规方法进行改进。基本的扫描都是从右向左，改进的地方就是利用了已经计算得到的suff[]值，计算现在正在计算的suff[]值。具体怎么利用，看下图：

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 |

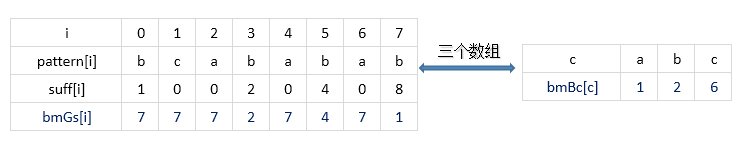
suff[4] = 4，这里f=4，g=0，当i=3是，这时suff[m-1=f+i]=suff[8]=3，而suff[3]=4，两者不相等，因为上一次的失配位置g可能会在这次得到匹配。

好了，这样解释过后，代码也比较简单：

|  |
| --- |
| **void suffix(char \*pattern, int m, int suff[]) {**  **int f, g, i;**  **suff[m - 1] = m;**  **g = m - 1;**  **for (i = m - 2; i >= 0; --i) {**  **if (i > g && suff[i + m - 1 - f] < i - g)**  **suff[i] = suff[i + m - 1 - f];**  **else {**  **if (i < g)**  **g = i;**  **f = i;**  **while (g >= 0 && pattern[g] == pattern[g + m - 1 - f])**  **--g;**  **suff[i] = f - g;**  **}**  **}**  **}** |

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：

[](http://jbcdn2.b0.upaiyun.com/2013/12/BM-math16.png)

**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：

|  |
| --- |
| **for(i = 0; i < m - 1; i++)**  **{**  **bmBc[pattern[i]] = m - 1 - i;**  **}** |

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

|  |
| --- |
| **#include <stdio.h>**  **#include <string.h>**  **#define MAX\_CHAR 256**  **#define SIZE 256**  **#define MAX(x, y) (x) > (y) ? (x) : (y)**  **void BoyerMoore(char \*pattern, int m, char \*text, int n);**  **int main()**  **{**  **char text[256], pattern[256];**  **while(1)**  **{**  **scanf("%s%s", text, pattern);**  **if(text == 0 || pattern == 0) break;**  **BoyerMoore(pattern, strlen(pattern), text, strlen(text));**  **printf("\n");**  **}**  **return 0;**  **}**  **void print(int \*array, int n, char \*arrayName)**  **{**  **int i;**  **printf("%s: ", arrayName);**  **for(i = 0; i < n; i++)**  **{**  **printf("%d ", array[i]);**  **}**  **printf("\n");**  **}**  **void PreBmBc(char \*pattern, int m, int bmBc[])**  **{**  **int i;**  **for(i = 0; i < MAX\_CHAR; i++)**  **{**  **bmBc[i] = m;**  **}**  **for(i = 0; i < m - 1; i++)**  **{**  **bmBc[pattern[i]] = m - 1 - i;**  **}**  **/\* printf("bmBc[]: ");**  **for(i = 0; i < m; i++)**  **{**  **printf("%d ", bmBc[pattern[i]]);**  **}**  **printf("\n"); \*/**  **}**  **void suffix\_old(char \*pattern, int m, int suff[])**  **{**  **int i, j;**  **suff[m - 1] = m;**  **for(i = m - 2; i >= 0; i--)**  **{**  **j = i;**  **while(j >= 0 && pattern[j] == pattern[m - 1 - i + j]) j--;**  **suff[i] = i - j;**  **}**  **}**  **void suffix(char \*pattern, int m, int suff[]) {**  **int f, g, i;**  **suff[m - 1] = m;**  **g = m - 1;**  **for (i = m - 2; i >= 0; --i) {**  **if (i > g && suff[i + m - 1 - f] < i - g)**  **suff[i] = suff[i + m - 1 - f];**  **else {**  **if (i < g)**  **g = i;**  **f = i;**  **while (g >= 0 && pattern[g] == pattern[g + m - 1 - f])**  **--g;**  **suff[i] = f - g;**  **}**  **}**  **// print(suff, m, "suff[]");**  **}**  **void PreBmGs(char \*pattern, int m, int bmGs[])**  **{**  **int i, j;**  **int suff[SIZE];**  **// 计算后缀数组**  **suffix(pattern, m, suff);**  **// 先全部赋值为m，包含Case3**  **for(i = 0; i < m; i++)**  **{**  **bmGs[i] = m;**  **}**  **// Case2**  **j = 0;**  **for(i = m - 1; i >= 0; i--)**  **{**  **if(suff[i] == i + 1)**  **{**  **for(; j < m - 1 - i; j++)**  **{**  **if(bmGs[j] == m)**  **bmGs[j] = m - 1 - i;**  **}**  **}**  **}**  **// Case1**  **for(i = 0; i <= m - 2; i++)**  **{**  **bmGs[m - 1 - suff[i]] = m - 1 - i;**  **}**  **// print(bmGs, m, "bmGs[]");**  **}**  **void BoyerMoore(char \*pattern, int m, char \*text, int n)**  **{**  **int i, j, bmBc[MAX\_CHAR], bmGs[SIZE];**  **// Preprocessing**  **PreBmBc(pattern, m, bmBc);**  **PreBmGs(pattern, m, bmGs);**  **// Searching**  **j = 0;**  **while(j <= n - m)**  **{**  **for(i = m - 1; i >= 0 && pattern[i] == text[i + j]; i--);**  **if(i < 0)**  **{**  **printf("Find it, the position is %d\n", j);**  **j += bmGs[0];**  **return;**  **}**  **else**  **{**  **j += MAX(bmBc[text[i + j]] - m + 1 + i, bmGs[i]);**  **}**  **}**  **printf("No find.\n");**  **}** |