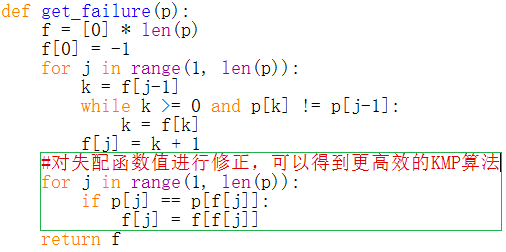
Python算法之旅（第11期）

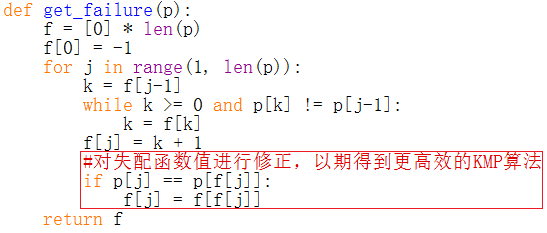
上期回顾：

描述：写出修正后的KMP算法失效函数。

算法分析：只遍历按照基本算法求出的失配函数，当p[j] == p[f[j]]时，进行f[j] = f[f[j]]的修正。



上期思考题中给出的那段颇具迷惑性的代码，看上去和正解差不多：



错误之处在于优化过早，当p[j]==p[0]时，过早地将f[j]修正成-1，致使f[j+1]得不到正解。例如当p = "AABB"时，就得不到正解。

还有一种与KMP算法类似的字符串模式匹配算法，那就是Boyer-Moore算法。与KMP算法不同的是，BM算法是基于后缀匹配的，后缀匹配就是模式串从右到左开始比较，但模式串的移动还是从左到右的。

此外，BM算法不仅预先分析模式串的特征（好后缀规则），还要预先分析目标串各个字符与模式串的关系（坏字符规则），生成好后缀列表和坏字符字典，这正是BM算法比KMP算法更高效的原因。

题目：字符串模式匹配算法之BM算法

难度：4星 有趣：3星 有用：3星

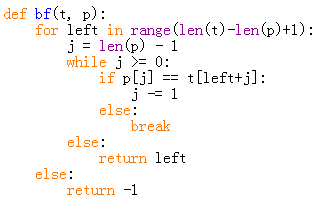
分类：字符串，模式匹配，打表

描述：设有目标串T(target)和模式串P(pattern)，模式匹配是指在目标串T中找到一个与模式串P相等的子串。模式匹配成功是指在目标串T中找到了一个模式串P。

算法分析：

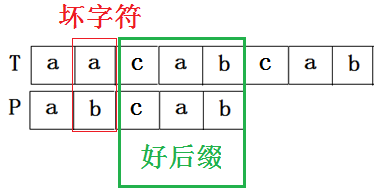
**一．简单粗暴的BF算法**

为了便于理解BM算法，我们把简单字符串模式匹配算法也写成后缀匹配形式。我们使用变量left指向当前T与P的左边界对齐的位置，j指向当前比较的位置，然后从右向左依次匹配，直至串P中的每个字符依次和串T中的一个连续的字符序列（即匹配子串）相等，则称匹配成功，返回left；否则成匹配不成功，返回-1。



**二．高效的BM算法**

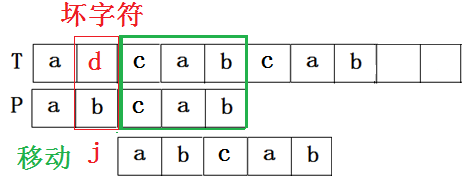
字符串匹配的关键在于如何正确而高效地移动模式串，Boyer-Moore为了做到这点定义了两个规则：坏字符规则和好后缀规则，下图给出了定义：



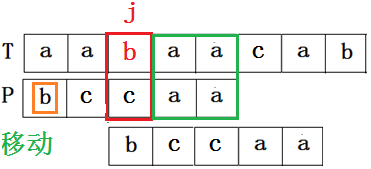
下面分别对利用坏字符规则和好后缀规则移动模式串进行介绍：

**坏字符规则**

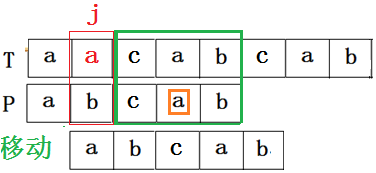
1.如果模式串不包含坏字符，则直接将模式串移动到坏字符的下一个字符（j+1）处：



2. 如果模式串包含坏字符，且最右侧的坏字符在位置j的左侧，则将模式串最右侧的坏字符与目标串的坏字符对齐：



3. 如果模式串包含坏字符，且最右侧的坏字符在位置j的右侧，因为此时不可能将模式串左移来与坏字符对齐，故选择将模式串右移1位：



我们已经知道了坏字符移动规则，现在分析具体的编码方法：

前面给出的代码中我们用left和j来指示位置，由于BM算法是基于后缀匹配的，我们不能像KMP算法中那样直接改变j的值，而是先求出位移量span，再移动left += span。

span是多少呢？显然它应该是j到模式串最右侧坏字符的距离。即：span = j - bc(badchar)。

当模式串包含坏字符时，bc(badchar)就是其下标；如果不包含坏字符呢？bc(badchar)又该是多少？在KMP算法中我们用f[k]= -1,表示k回溯到了最左端，今天可以采用类似的方法处理模式串不包含坏字符的情形，即设bc(badchar) = -1。

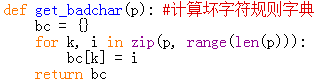
仔细计算一下，我们就会发现，这种情况下，刚好能够让整个模式串调到坏字符的下一位。

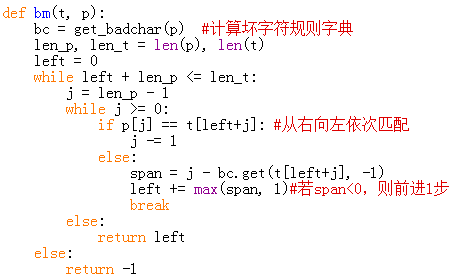
接下来的问题是如何计算bc(badchar)？

Python语言天生就适合解决这个问题，我们可以创建一个字典，以模式串中元素的值为键，下标为值。因为只需考虑模式串中最右侧的坏字符，故对键相同的元素，可以用较大的下标覆盖较小的下标值。

在计算span时，我们调用字典的get() 函数返回指定键的值，若模式串包含坏字符，可直接返回其值（最大下标），否则返回-1。用j减去get() 函数的返回值就得到span。

相关代码如下：





利用坏字符规则，可以使模式串向前移动的距离变大，但还不够好。

例如，当T = "aaaaaaaa"，P = "baaa"时，每次只能前进一步，算法的执行流程完全等同于BF算法。效率低下的原因在于该算法完全无视了之前存在部分匹配的事实。所以仅依靠坏字符规则是不够的，我们还得看看好后缀规则。

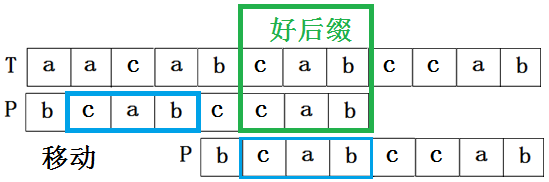
**好后缀规则**

在有了“坏字符”的基础上我们还要加上 “好后缀”规则。所谓“好后缀”就是从后往前匹配部分的后缀，即模式串中与“坏字符”对应位置右侧的后缀。

与KMP算法的失配函数类似，我们可以根据模式串本身计算移动函数，利用“好后缀”对应的移动函数值来决定模式串的位移量，以达到尽可能多地匹配部分子串的目标。移动的情况有三种:

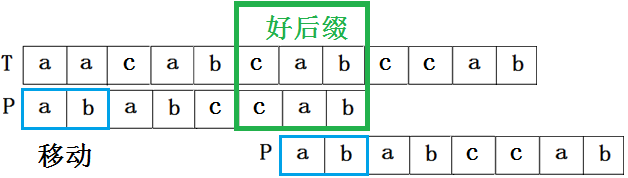
1. 模式串中有与“好后缀”完全匹配的子串。

此时只需将最近的匹配子串和好后缀重合就行了，位移量就是好后缀到其最近匹配子串的距离。

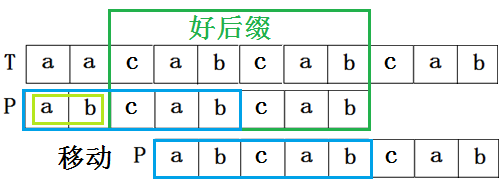


2. 模式串中没有与“好后缀”完全匹配的子串，但存在一些子串与“好后缀”的后缀匹配，因为这些子串与模式串的左端对齐，我们称其为“前缀”，其中长度最大的称为“最长前缀”。

此时只需将“最长前缀”和“好后缀”右端重合就行了，位移量就是“好后缀”右端到“最长前缀”的距离。

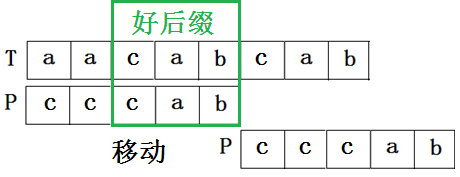


注意有时“最长前缀”和“好后缀”可能会出现部分字符重合的情形，此时移动规则是一样的。在本例中好后缀”cabcab”有两个前缀”ab”和”abcab”，其中“最长前缀”是”abcab”。



3. 模式串中没有与“好后缀”完全匹配的子串，也不存在任何“前缀”。

此时只需将整个模式串移到自己右侧就行，即位移量为模式串的长度。



与KMP算法的失配函数类似，我们可以根据模式串本身计算移动函数，建立与各个“好后缀”对应的“移动距离表”，我们用列表gs来存储“移动距离表”，其中gs[j]表示当位置j与目标串的“坏字符”对齐时，模式串右移的距离。

为了方便计算gs[j]的值，我们预先生成一个“前缀”长度表suf（这个技巧称为“打表”，例如在处理与素数有关的问题时，人们预先生成一个素数表），其中suf[i]表示以i为右边界，与模式串后缀匹配的最大长度。

例如，模式串P = "baaa"时，suf = [0, 1, 2, 4]；

模式串P = "cbcab"时，suf = [0, 1, 0, 0, 5]；

模式串P = "abcabcab"时，suf = [0, 2, 0, 0, 5, 0, 0, 8]；

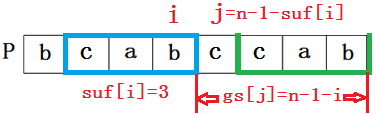
有了suf表之后，我们就可以构建gs表了。

构建gs列表分为三种情况，分别对应上述移动模式串的三种情况:

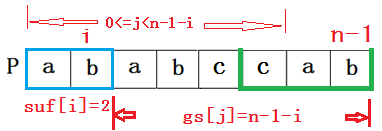
1. 模式串中有与“好后缀”完全匹配的子串。

图示中模式串的长度为n，i是匹配子串的右边界，j与目标串的“坏字符”对齐。

如下图gs[8-1-3] = gs[4] = 8-1-3 = 4。



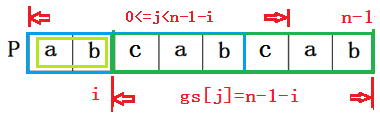
1. 模式串中没有完全匹配子串，但是存在“前缀”。

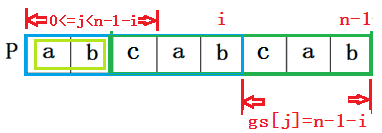


若存在多个“前缀”，则需多次计算gs[j]的值，并取最小值作为gs[j]的最终值。

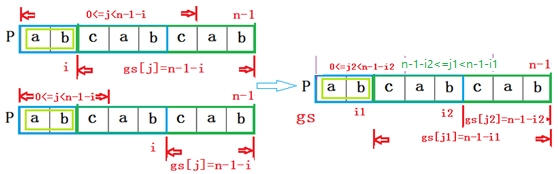
如下图，好后缀”cabcab”有两个前缀”ab”和”abcab”，从左向右遍历模式串，当i = 1时，找到前缀”ab”，利用它来计算各个gs[j]的值，对于所有的0<=j<n-1-i，均有gs[j]=n-1-I，即gs[0:6] =[6,6,6,6,6,6]；

同理，当i = 4时，找到前缀” abcab”，得到gs[0:3] = [3,3,3]。

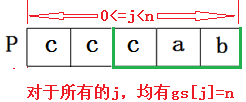




因为前缀越长，移动的距离就越短，本着“宁可慢点，不能错过”的原则，每当在位置j遇到坏字符的时候，我们应该选择移动距离短的gs[j]值。即在本例中，虽然根据前缀”ab”得到了gs[0:6] 的值，但是后来根据前缀”abcab”得到gs[0:3]的新值要覆盖旧值，二者共同作用的结果是gs[0:6] =[3,3,3,6,6,6]。



1. 模式串中既没有完全匹配子串，也不存在任何“前缀”。

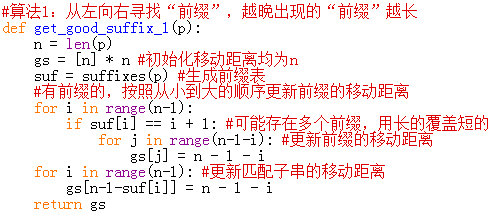


那么，在计算列表gs时，如果同时存在“匹配子串”和“前缀”，我们优先考虑哪个呢？

答案是“宁可慢点，不能错过”。对于上述三种情况，我们优先考虑匹配子串，其次考虑“前缀”，最后才是移动整个模式串。

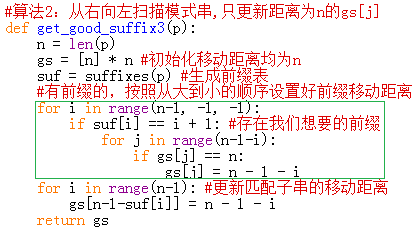
从代码实现的角度来说就是先默认初始化移动距离均为n，再根据“前缀”规则更新gs[j]的值，最后根据“匹配子串”规则更新gs[j]的值。

根据上述分析，我们得到最直接的代码如下：

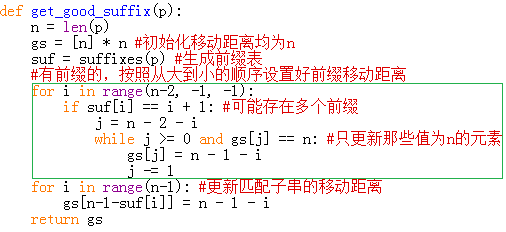


算法1是从左向右扫描模式串，这样越晚出现的“前缀”越长，对应移动距离越小，所以后来的gs[j]要更新先前计算出的gs[j]。

网络上比较流行的一种写法是从右向左扫描模式串，这样越早出现的“前缀”越长，对应移动距离越小。对于后来出现的“前缀”，我们只用它更新距离为n的gs[j]的值，这样效率略有提升。代码如下：



算法2看似效率提高了，但还不够，原因是它对g[0:n-1-i]都进行了判断，但之前已有较长“前缀”确定了对应范围内g[j]的值，而且这个范围是连续的，所以一旦遇到gs[j] != n，就没有必要再向左侧扫描了。改进后的代码如下：



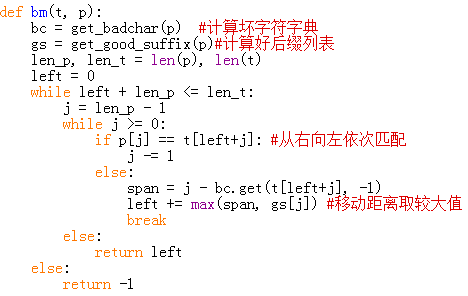
由此我们就可以根据“好后缀”规则，生成模式串的gs表了。

例如，模式串P = "baaa"时，gs = [4, 1, 2, 3]；

模式串P = "cbcab"时，gs = [5, 5, 5, 3, 1]；

模式串P = "abcabcab"时，gs =[3, 3, 3, 6, 6, 6, 8, 1]；

有了gs表，再结合“坏字符”字典bc，我们就可以写出完整的BM算法程序了。

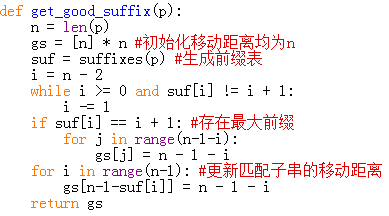


怎么样，BM算法也不是想象中的的那么难吧？

课后思考：

在计算gs表时，有一个比较有迷惑性的方法，看上去非常简洁高效，而且许多时候也能获得正确答案。

对应代码如下：



聪明的你请仔细思考，看看这段代码问题出在哪里？会带来什么危害呢？

另外，如果你有更 Pythonic（优雅的、地道的、整洁的）代码，或者与本文不同的算法思路和代码实现，请你一定留言或联系我，让我们一起讨论，共同进步。