Python算法之旅（第10期）

题目：字符串模式匹配算法之KMP算法

难度：4星 有趣：3星 有用：3星

分类：排序

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。

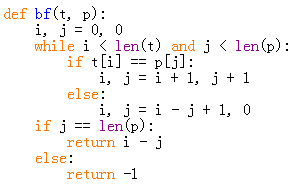
算法分析：

一．简单粗暴的BF算法

简单字符串模式匹配算法（也就是BF算法）的基本思想是：从目标串T的起始比较位置pos开始（在后面的案例中，我们默认pos = 0），和模式串P的第一个字符比较之，若匹配，则继续逐个比较后继字符，否则从串T的下一个字符起再重新和串P的字符比较之。

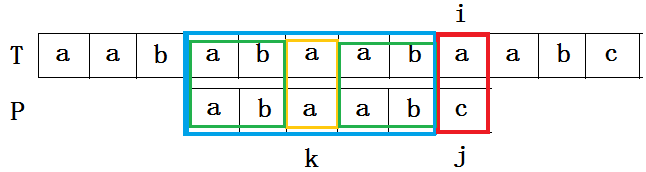
依此类推，直至串P中的每个字符依次和串T中的一个连续的字符序列（即匹配子串）相等，则称匹配成功，返回该匹配子串的首字符下标；否则匹配不成功，返回-1。

对应Python代码如下：



我们发现，在某一轮比较中，一旦出现字符失配（即T[i] != P[j]），则需将i和j回溯，其中i回溯至i = i - j + 1，j回溯至j = 0。

这样产生了很多不必要的比较，例如（例1）：



T = "aababaabaabc"

P = "abaabc"

在第4轮比较中，T3T4T5T6T7 == P0P1P2P3P4，我将其简写为T[3…7] == P[0…4]，但T[8] != P[5]，出现字符失配，需要将i和j回溯，使得i = 4，j = 0。

而在第4轮比较中，我们已经得到了T[6…7] == P[3…4]，又P[0…1] == P[3…4]，相当于T[6…7]和P[0…1]已经间接地比较过，而且字符匹配了，我们无需进行从i =6，j=0开始的重复比较。

实际上，当T[8] != P[5]，即在i = 8，j = 5处出现字符失配时，我们无需将i回溯，只需将j回溯至f[j]（此时f[5] = 2）处即可。即当T[8] != P[5]时，我们可以跳过比较T[6…7]和P[0…1]（因为它们已经间接地比较过了），直接比较字符T[8]和P[2]，这样可以省去很多不必要的回溯和比较，时间复杂度达到O（lenT+lenP）。这就是KMP算法的核心思想。

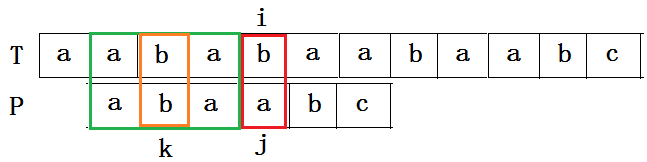
二．高效的KMP算法

KMP是三位大牛：D.E.Knuth、J.H.Morris和V.R.Pratt同时发现的。

我在上文提到当T[i] != P[j]时，我们无需将i回溯，只需将j回溯至f[j]处即可。我们称f[j]为模式串P下标j的失配函数。

失配函数的值f[j]是指当T[i] != P[j]时，接下来与T[i]进行比较的模式串P的元素下标。如上面的例子，当T[8] != P[5]时，因为T[6…7] == P[3…4] == P[0…1]，我们可以跳过比较T[6…7]和P[0…1]，直接比较字符T[8]和P[2]，所以f[5] = 2。

如果你对失配函数还不太理解，我再举一些例子。



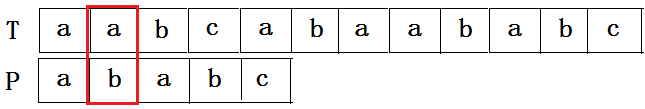
仍然以上面提供的目标串T和模式串S为例，当出现T[1…3] == P[0…2]，但T[4] != P[3]时，若采用BF算法，则需要将i和j回溯，使得i = 2，j = 0。

而采用KMP算法，则i无需回溯，j也只需回溯至j = f[j]。那如何得知f[j]的值呢？观察模式串P，我们发现P[2] == P[0]，因为T[3] == P[2]，所以T[3] == P[0]，相当于我们已经间接地比较过T[3]和P[0]了，无需重复比较，接下来可以直接比较T[4]和P[1]，所以f[3] = 1。

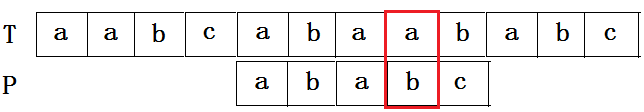
再看一个简单的例子（例2）：

T = "aabcabaababc"

P = "ababc"



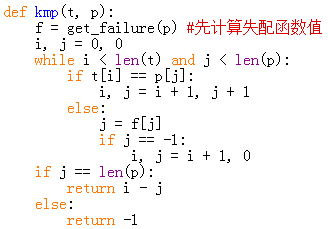
当出现T[0] == P[0]，但T[1] != P[1]时，要保证i不变，必须将j回溯至j = 0，然后比较T[1]和P[0]，所以f[1] = 0。



同样的，当出现T[4…6] == P[0…2]，但T[7] != P[3]时，因为P[2] == P[0]，所以T[6] == P[0]，相当于我们已经间接地比较过T[6]和P[0]了，无需重复比较，接下来可以直接比较T[7]和P[1]，所以f[3] = 1。

那么，如果模式串P的第一个元素就不匹配，即T[i] != P[0]又该怎么办呢？j已经最小，没办法再往前回溯了，下一次比较必须使i自增1。这是一种特殊的情况，考虑到Python语言中的数组下标从0开始，为了表示区别，我们设f[0] = -1。很明显当f[j] != -1时，在进行下一次比较之前，我们无需改变i的值；而当f[j] == -1时，在进行下一次比较之前，必须先使i自增1。

据此，我们可以写出KMP算法的代码。它在形式上和BF算法极为相似，不同之处仅在于：当匹配过程中产生“失配”时，目标串T指示标志i不变，模式串P指示标志j回溯至f[j]所指示的位置，当且仅当j回溯至最左端（即f[j] == -1）时，使j = 0，i自增1。



三．计算失配函数

先结合上面的例子理解失配函数的意义。

设在目标串T中查找模式串P，若T[i] != P[j]，则将j回溯至失配函数值f[j]处，那f[j]可以取到哪些值呢？

1. f[0]= -1，表示T[i]和P[0]间接比较过了，且T[i] != P[0]，接下来比较T[i+1]和P[0]；

② f[j] = k，其中0 < k < j，表示T[i]之前的k个字符与P中的前k个字符已经间接比较过了，且P[0…k-1] == P[j-k…j-1] == T[i-k…i-1]，接下来比较T[i]和P[k]。

③ f[j] = 0，表示比较过程中产生了不相等，接下来比较T[i]和P[0]；

除了上述三种情况，f[j]不可能取到其他值。

那么如何求解失配函数f[j]的值呢？

从上述讨论可见，失配函数f[j]的值仅取决于模式串P本身，与目标串T无关。

①f[0]= -1：考虑到数组下标从0开始，首字符的失配函数值规定为-1；

1. f[j] = k：若P[0…k-1] == P[j-k…j-1]，其中0 < k < j。
2. f[j] = 0：除①②的其他情况。

如：若P ="ababc"，则对应失配函数f = [-1, 0, 0, 1, 2]；

若P ="abcabb"，则对应失配函数f = [-1, 0, 0, 0, 1, 2]；

  若P ="abcaabcab"，则对应失配函数f = [-1, 0, 0, 0, 1, 1, 2, 3, 4]；

搞清楚了计算原理，如何用代码实现呢？

先给出两个概念：若存在0 <= k < j，且使得P[0…k] == P[j-k…j]的最大整数k，我们称P[0…k]为串P[0…j]的前缀子串，P[j-k…j]为串P[0…j]的后缀子串。

从f[j]的定义出发，计算f[j+1]就是要在串P[0…j]中找出最长的相等的前缀子串P[0…k]和后缀子串P[j-k…j]，这个查找的过程实际上仍是一个模式匹配的过程，只是目标和模式现在是同一个串P。

我们可以用递推的方法求f[j+1]的值。

设已有f[j] = k，则有0 < k < j，且P[0…k-1]==P[j-k…j-1]。接下来：

若P[k]==P[j]，则由f[j]的定义可知f[j+1] = k + 1 = f[j] + 1；

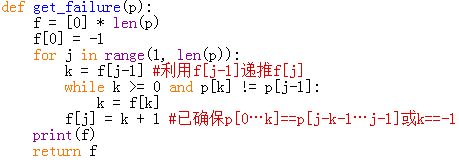
若P[k] != P[j]，则可以在前缀子串P[0…k]中寻找使得P[0…h-1] == P[k-h…k-1]的h，这时存在两种情况：

①找到h，则由f[j]的定义可知f[k] = h，故P[0…h-1]==P[k-h…k-1]==P[j-h…j-1]，即在串P[0…j]中找到了长度为h的相等的前缀子串和后缀子串。

这时若P[h]==P[j]，则由f[j]的定义可知f[j+1] = h + 1 = f[k] + 1 = f[f[j]] + 1；

若P[h] != P[j]，则再在串P[0…h]中寻找更小的f[h]。如此递推，有可能还需要以同样的方式再缩小寻找范围，直到f[h] == -1才算失败。

②找不到h，这时f[k]==-1，即k已经回溯到k = f[k]= -1，所以f[j+1] = k + 1 = 0。



KMP算法的代码虽然简短，但是其原理却不好理解，如果一遍看不懂，请看两遍。结合图例多看几遍，自己手动计算一下模式P的失配函数，再手写代码，慢慢领会。

课后思考：

前面定义的失效函数在某些情况下尚有缺陷。

例如，当模式串P = "aaaaaab"时， f = [-1, 0, 1, 2, 3, 4, 5]。若T[i] != P[6]，因为f[6] = 5，所以下一步要将T[i]和P[5]比较；依此类推还要比较P[4]，P[3]，…，P[0]。实际上，因为它们都相等，所以当T[i] != P[6]时，可以直接比较T[i]和P[0]。

也就是说，若按上述定义得到f[j] = k，且P[j] == P[k]时，则当T[i] != P[j]时，不需要再比较T[i]和P[k]，可以直接比较T[i]和P[f[k]]，即此时的f[j]应该等于f[k]。

由此我们可以在原来计算失效函数算法的基础上加上一些语句，对失效函数值进行修正，以得到更高效的KMP算法。

聪明的你请想一想，这些修正失效函数的语句该怎么写？又是加在哪个位置呢？

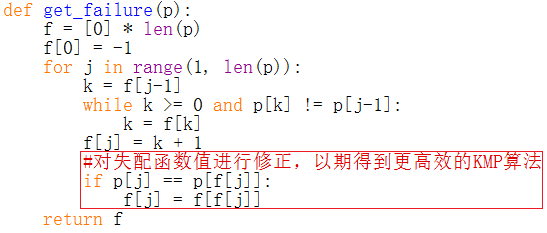
提示，根据修正后的失效函数，我们得到以下示例：

若P ="aaaaaab"，则对应失配函数f = [-1, -1, -1, -1, -1, -1, 5]；

若P ="abcabb"，则对应失配函数f = [-1, 0, 0, -1, 0, 2]；

若P ="abcaabcab"，则对应失配函数f = [-1, 0, 0, -1, 1, 0, 0, -1, 4]；

另外给一段颇具迷惑性的代码，很多时候它都能正确工作，但却是错的。你知道它错在哪里吗？



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。