字符串匹配

暴力匹配
与
KMP 算法

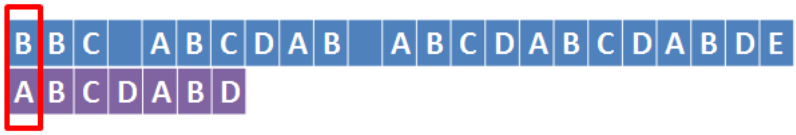
就按照网上那个经典的例子来讲解吧：

我们要从字符串“BBC ABCDAB ABCDABCDABDE”里面查找它是否存在一个字符串“ABCDABD”

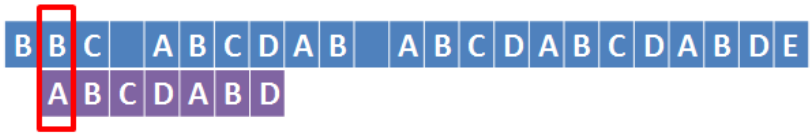
暴力匹配

先解释一下暴力匹配的方法：

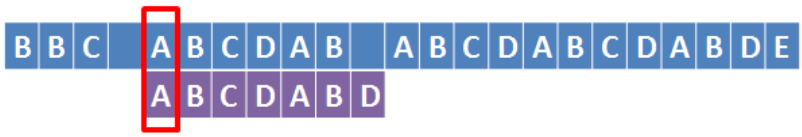
首先将两个字符串左边对齐，然后匹配第一个字符：



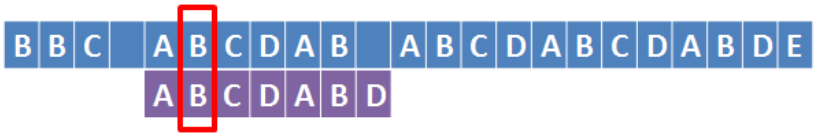
如果不能匹配成功，就把下面的字符串往右移动一格，继续匹配：

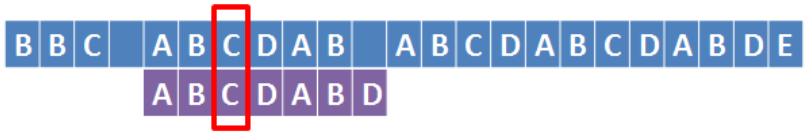


一直到第一个字符匹配成功：

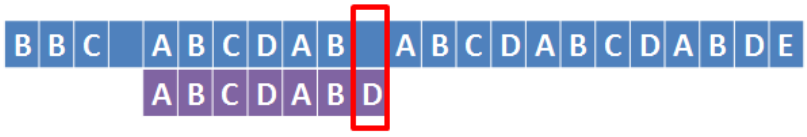


这时候我们就应该匹配第二个字符，一直到有某个字符不匹配，或者所有字符都匹配成功：

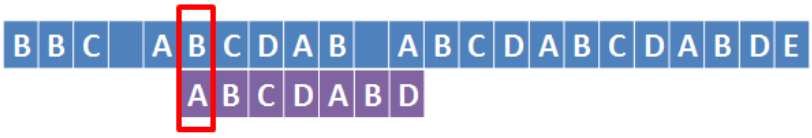




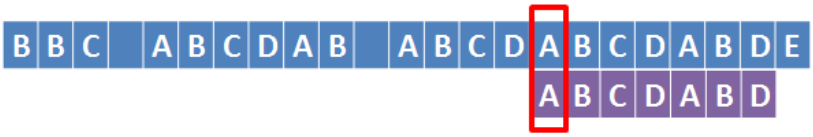
...

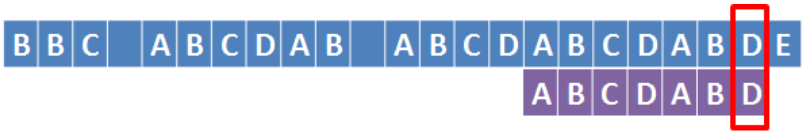


这时候到了匹配不成功的字符，在暴力匹配的情况下，我们是将下面的字符串往右移动一格，变成：



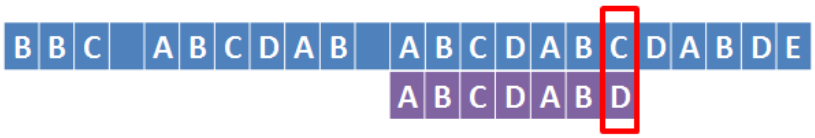
然后就重复上面的最前面的操作，一直到匹配成功，或者下面的字符串移动到了上面字符串的最后。



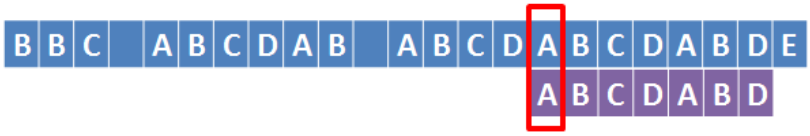


KMP 算法

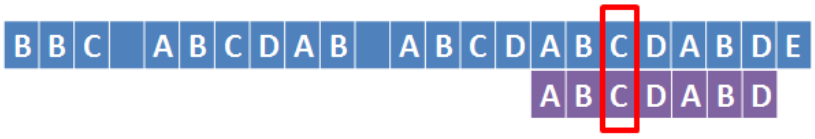
首先我们看一下前面暴力匹配会遇到的这种情形：



我相信你一眼就能看出来，其实我们下一步可以把下面那个字符串直接移动成这样：



你可能觉得，这不是跟前面暴力匹配一样嘛，一个个格子移动过来效率也低不了多少的嘛...是的，如果你是这样移动，那确实效率相差不大，那么接下来，看看下面这种移动方式：



这里跟上面唯一的不同就是红色框的位置还是在上面的 C 字符那里，这个我相信你一眼看上去还是可以接受的，这样子效率会高很多么？

这里跟前面暴力匹配的区别有两个：

1、 上面匹配不成功，直接往右移动 n 个位置，使得它能够匹配上前面的 AB

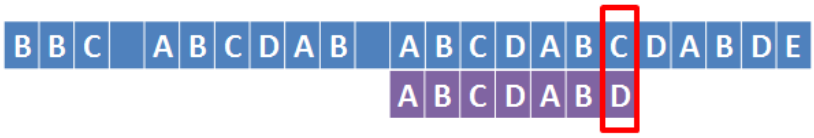
2、 前面已经匹配过的 AB，不需要再次匹配，

那么假设下面那个字符串，C 前面有很多个字符呢？是不是就有很多字符不用匹配了？

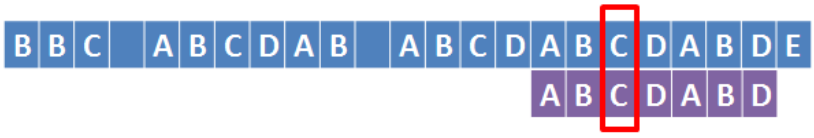
这就是 kmp 算法的基本思想。

那么问题来了，为什么可以这样？kmp 算法的基本思想到底是个什么东西？

我们仔细观察一下这个情况：



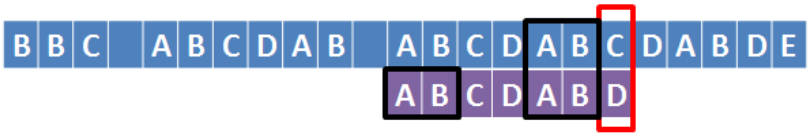
可以发现，在红色框不匹配的时候，红色框前面的 AB 是可以与下面那个字符串开头的 AB 匹配的，所以我们可以直接把下面那个字符串的 AB 移动到这里，然后继续匹配后面的字符，也就是：



那么问题又来了，我怎么知道可不可以往右直接跳过格子，如果可以的话，我到底跳过几个格子呢？

所谓的跳过几个格子，无非就是匹配到目前，虽然不能成功匹配，但是我知道前面有几个字符是跟我的开头完全匹配的，这时候我就可以直接把它移动到这个匹配的地方。

就拿这张图来说：

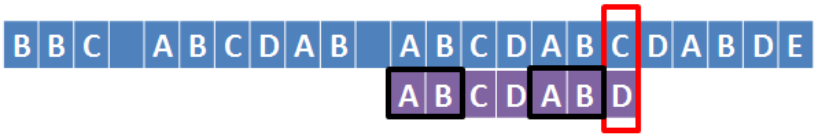


黑色框告诉我们，如果当前红色框匹配不成功，下次匹配可以直接从这里开始，因为两个黑色框里面是相同的，不需要重复匹配。（这里从图中可以看出，我们可以直接移动的格数为：红框中的 D 的下标 － 黑色框的字符数 ＝6－2＝4）

理解了这个基本思想之后，接下来我们就该考虑，我们要求解这个问题还需要什么东西，红框中 D 的下标，这个我们在匹配的时候就有的，而黑框中的字符个数，这个我们是不知道的，所以问题就变成了，如何求解这个黑框中的字符个数。

首先，它跟上面那个字符串一点关系都没有，也就是，我们只需要在下面那个字符串上面动手脚就可以了。

这里我先把结论写出来，如果你不能理解这个结论，那么你可以多用几个字符串匹配试试。



我们只看下面那个字符串，当我们匹配到 D 的时候，这个 D 前面的 AB 肯定是已经匹配了的，所以如果 D 不匹配，我们知道 D 前面的 AB 肯定可以跟最前面 AB 匹配，所以可以移动过来。

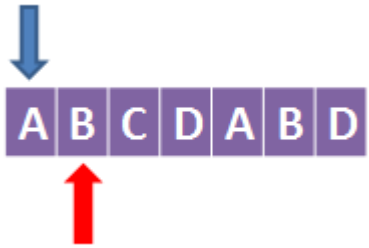
接下来我们把它抽象成普遍的情况：

当我们匹配到第 i 个字符时，如果它不匹配，并且这个字符前面的 k 个字符与字符串最前面的 k 个字符是相同的，那么我们可以直接把下面那个字符串往右移动(i-k)个格子，对应上面的例子，i＝6，k＝2，注意这里的 k＜i，不然你 k＝i 的话不就是把前面所有字符都框进来了嘛，那样的话，你等下要移动 i－k 个格子不就是相当于没有移动了嘛...

所以其实我们的问题就是求解下面那个字符串中，每个位置的 k 值，因为每个位置都有可能匹配不成功，所以我们需要把每个位置的 k 值都求出来备用。

你可以自己思考一下如果是你，你会怎么求...

接下来讲一下怎么求解：



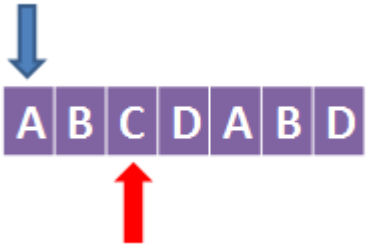
上面蓝色的箭头表示从头匹配到哪个字符，下面红色的箭头表示扫描到哪个字符。

因为第 0 个字符不需要扫描，它前面没有字符，所以也就没有 k 值...因为 k < i，所以我们给它的 k 值是-1。

接下来扫描第 1 个字符，B，它前面只有一个字符，因为第 0 个字符是不能被框进那个黑色框的，所以它的 k 值为 0。

前面两种属于特殊情况，接下来从第三个字符开始就是循环扫描了：

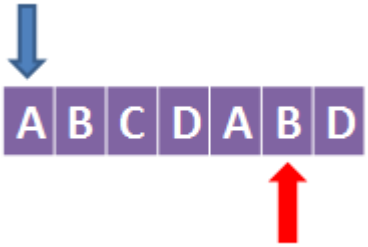
对于第 2 个字符，C：



查看它的前一个字符是否与蓝色箭头 t 指向的字符相同，如果相同，则 t++，该字符的 k 值等于自增之后的 t，如果不相等，则 k 值为 0，并将蓝色箭头重置到第 0 个字符。

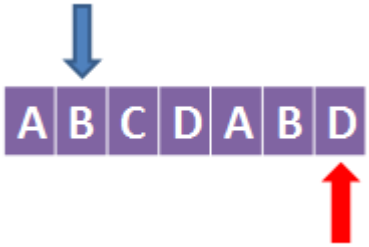
第 3 个和第 4 个字符都是一样的结果，k = 0，t = 0

接下来看第 5 个字符，B：



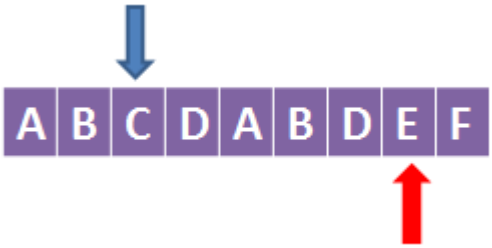
此时 B 前面的字符 A 等于蓝色箭头指向的 A，所以蓝色箭头往右移动一格：t++，k = 1

接下来看第 6 个字符，D：

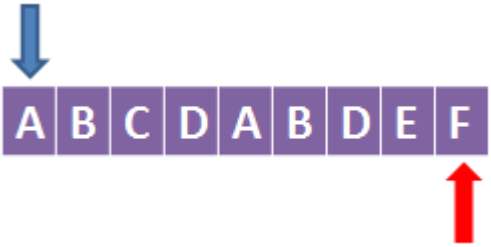


此时 D 前面的字符 B 等于蓝色箭头指向的 B，所以蓝色箭头往右移动一格：t++，k = 2

假设我们后面还有字符，假设是这样的：



此时由于前面的 D 不等于蓝色箭头指向的 C，所以重置蓝色箭头到最前面，t = 0，k = 0
于是就变成了：



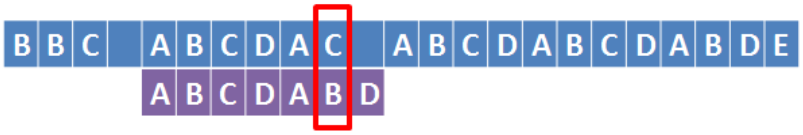
基本思路就是这样，后面这个实现的思路跟网上略有差别，网上一般是采取先包含红色箭头的字符，最后再整体将所有的 k 值往右移动一格，再给第一个位置的 k 值补-1，本质都是一样的。

现在我们把每个字符的 k 值都求出来了，匹配的时候就按照前面说的去匹配就好了，废话就不多说了...

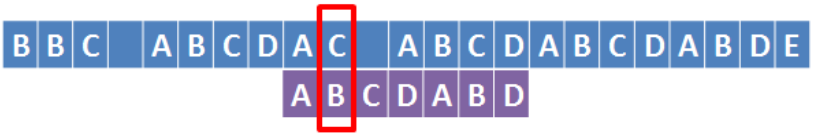
先看一下我们现在求出来的 k 值，也就是网上资料说的 next 数组：

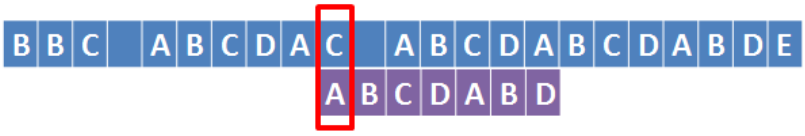
A	B	C	D	A	B	D	E	F
-1	0	0	0	0	1	2	0	0

事实上如果你做到上面这些，基本上已经能完成 kmp 的基本功能了，只是稍微有点可以优化的地方...注意这里我把原来的 B 改成了红框中的 C...



匹配到这时候，你觉得以你的智商，你会选择以下哪种情况作为你的下一步？



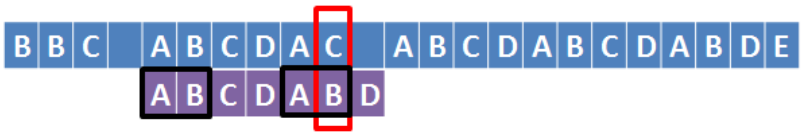


不管你选哪一个，如果两张图摆在这里给我看...反正我是会选第二个了...

那么区别在哪里呢？

最简单的区别就是下面的比上面的多移动了一格...

原因呢？



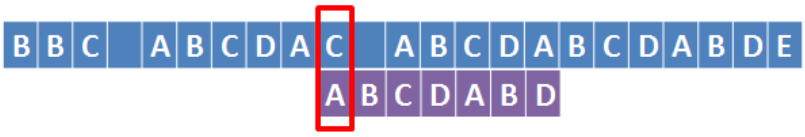
看着两个黑色框里面的字符是一样的，那么既然它不能匹配当前的 B，那我就算把 A 移动过来，它还是 B，所以依然是不能匹配的，这时候我们的下一步就是继续往右移动，那么能不能把这两步合起来？

这里的重点在于这个黑色框里面，刚刚我们求出来的 k 值是这样的，假设这个字符串叫 p：

A	B	C	D	A	B	D	E	F
-1	0	0	0	0	1	2	0	0

我们发现，如果在上面那个红色框匹配失败，然后移动，这时候，对着 A 的是第 0 个字符 A，对着 C 的是第 1 个字符 B，这时候就出现，刚刚我已经检测了 B 和 C 不匹配了，何必再检测一次呢...

因为我们的 p[5] = p[1] = B，因为 next[5] = 1，所以移动过来肯定是 p[1]对应着这个红框，所以移动过来就肯定会出现下一次的 p[1]跟 C 不匹配，那就可以直接把它移动成：



（为什么不把继续往后移动？因为你不知道上面那个字符是什么，这里我给的是 C，也许你遇到的是 A 呢...）

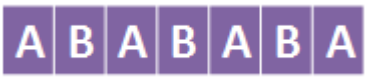
抽象来说：

当 p[i] = p[next[i]]的时候，next[i]可以直接用 next[next[i]]替代，这样一直循环直到两个字符不相等或者 next[i] = -1（实际上只要一次替换就可以了，后面会解释为什么）

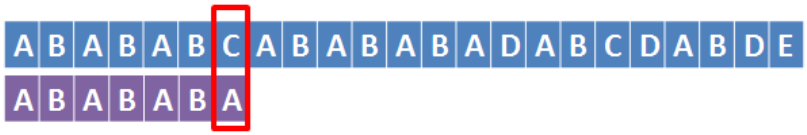
所以最终的 next 数组应该是这样的：

A	B	C	D	A	B	D	E	F
-1	0	0	0	-1	0	2	0	0

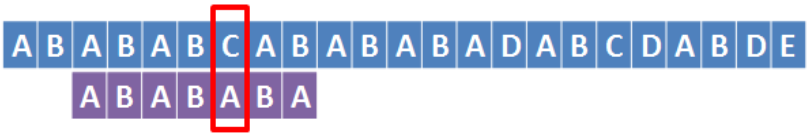
如果你觉得上面那句抽象看不懂的话...自己试着分析一下这个字符串的 next 数组应该是怎样的：

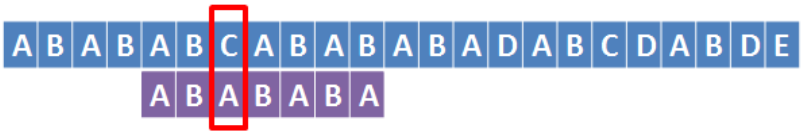


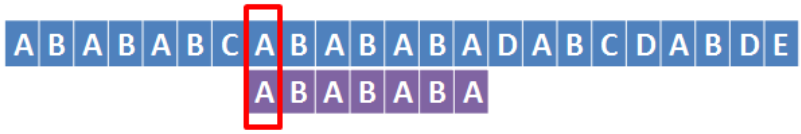
然后看看下面这种情况，下一步你会把这个字符串向右移动几格：



下面有 N 种情况供你选择：





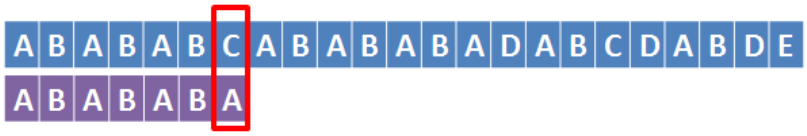


我相信正常人看完前面那么多图+废话...应该不会选前两种了...

现在问题是，到底是第三种还是第四种？

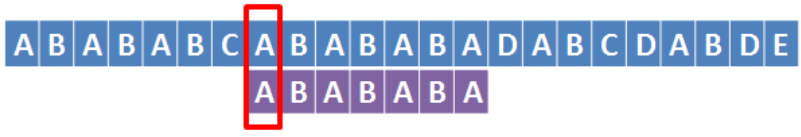
如果按照前面的思维，你可能觉得应该选第三种，因为你不知道上面那个字符串的红框里面的字符到底是不是我要的字符 A，所以我应该再匹配一次...

恩...好像说得很有道理的样子...可是...我把移动前的图复制下来，想想：



这时候你要移动，是不是就已经是知道了上面的 C 跟 A 是肯定不匹配的，而这里红框中的 A 又和下面字符串开头的 A 是相同的，所以是不是就不需要再匹配一次了？

所以是不是应该移动成这样了：



接下来我们看一下求出来的 next 数组应该是怎样的：
按照我们没有优化的时候，next 数组应该是这样的：

A	B	A	B	A	B	A
-1	0	0	1	2	3	4

但是我们看看优化的时候，一步一步是怎么求的（这个字符串我们叫做 p，取 pattern 之意）：

下标i	0	1	2	3	4	5	6
p[i]	A	B	A	B	A	B	A
next[i]	-1	0	0				
p[next[i]]	-	A	A				

如果不清楚这里的 next[2] = 0 是如何求出来的，请往前翻回去看。
这里我们看到了 p[2] = p[next[2]] = p[0] = A，所以我们让 next[2] = next[0] = -1：

下标i	0	1	2	3	4	5	6
p[i]	A	B	A	B	A	B	A
next[i]	-1	0	-1				
p[next[i]]	-	A	-				

接下来继续往后求，本来 B 的 next 值是 1：

下标i	0	1	2	3	4	5	6
p[i]	A	B	A	B	A	B	A
next[i]	-1	0	-1	1			
p[next[i]]	-	A	-	B			

但是我们看到了 p[3] = p[next[3]] = p[1] = B，所以我们让 next[3] = next[1] = 0（这里想想，需不需要再判断 p[3] = next[3] = p[0]是否成立？我们不是把 next[3]变成了 0 么？那假如我第 0 个字符就是 B 呢？）
那我告诉你，假如你第 0 个字符是 B，那么你第 1 个字符的 next 是不是在求解的时候就变成了 next[0] = -1 了，所以是不是就不需要循环赋值下去了？

下标i	0	1	2	3	4	5	6
p[i]	A	B	A	B	A	B	A
next[i]	-1	0	-1	0			
p[next[i]]	-	A	-	A			

接下来继续看下一个 A，本来应该是：

下标i	0	1	2	3	4	5	6
p[i]	A	B	A	B	A	B	A
next[i]	-1	0	-1	0	2		
p[next[i]]	-	A	-	A	A		

但是这里 p[4] = p[next[4]] = p[2] = A，所以我们让 next[4] = next[2] = -1，看到了吧，它直接就变到-1 那里去了，而不是先 next[4] = 2，然后再发现 p[next[2]] = p[0] = A 在把它变成-1：

下标i	0	1	2	3	4	5	6
p[i]	A	B	A	B	A	B	A
next[i]	-1	0	-1	0	-1		
p[next[i]]	-	A	-	A	-		

后面就同理了，不多赘述了，直接贴结果：

下标i	0	1	2	3	4	5	6
p[i]	A	B	A	B	A	B	A
next[i]	-1	0	-1	0	-1	0	-1
p[next[i]]	-	A	-	A	-	A	-