字符串模式匹配扩展*

学习目标

了解实现字符串匹配的其它算法

了解KMP， BM，KR和Sundays的各自算法特点

BM算法*

BM（Boyer-Moore）算法提出的动机与KMP算法类似，本质都是通过某种规律，在模式串和目标串匹配到某一个位置出现失配的情况时，跳过一些肯定不会匹配的情况，将模式串往后多滑动几位。

KMP算法与BM算法区别：

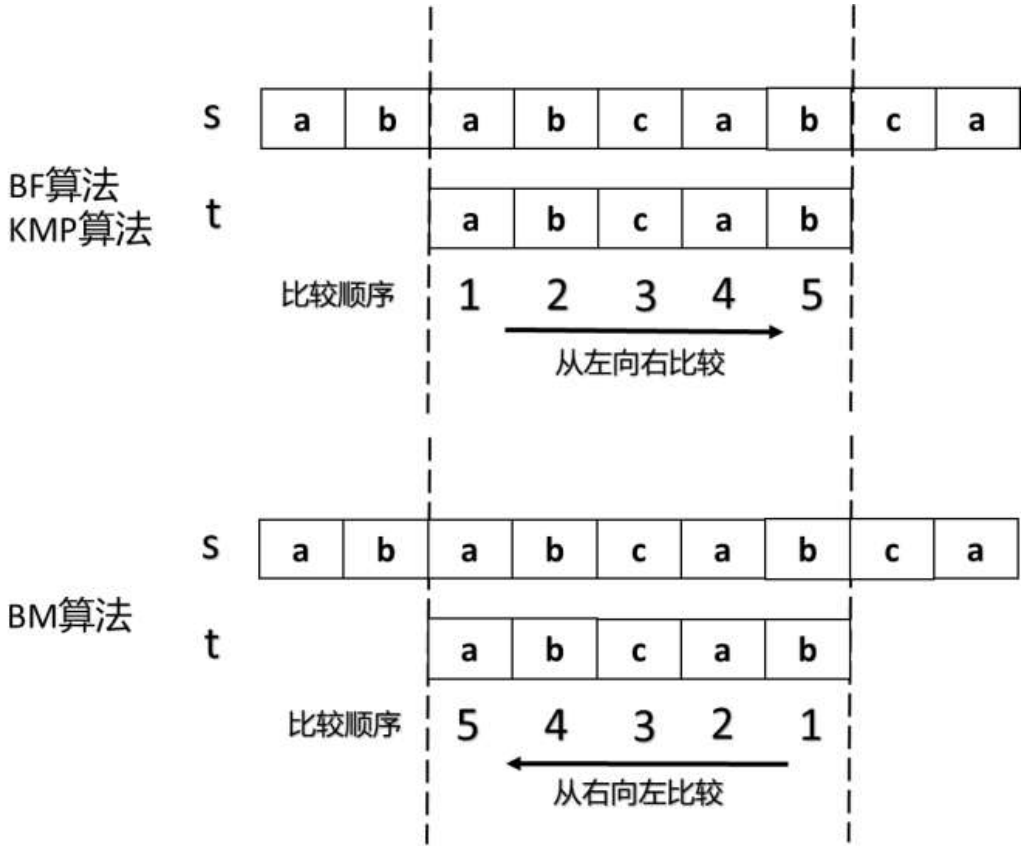
KMP算法：基于模式串前缀子串的最长公共前后缀长度来决定将模式串向后滑动多少

BM算法：使用“坏字符规则(Bad Character Rule，简称BC)”以及“好后缀规则(Good Suffix Rule，简称GS)”这两种规律来决定模式串向后滑动的位数，从而显著降低字符串模式匹配的复杂度，在效率上更优于KMP算法。

BM算法*

不同于朴素模式匹配算法以及KMP算法，BM算法是一种基于后缀匹配的模式匹配算法，即使用从右向左的比较策略来进行模式串和目标串的匹配

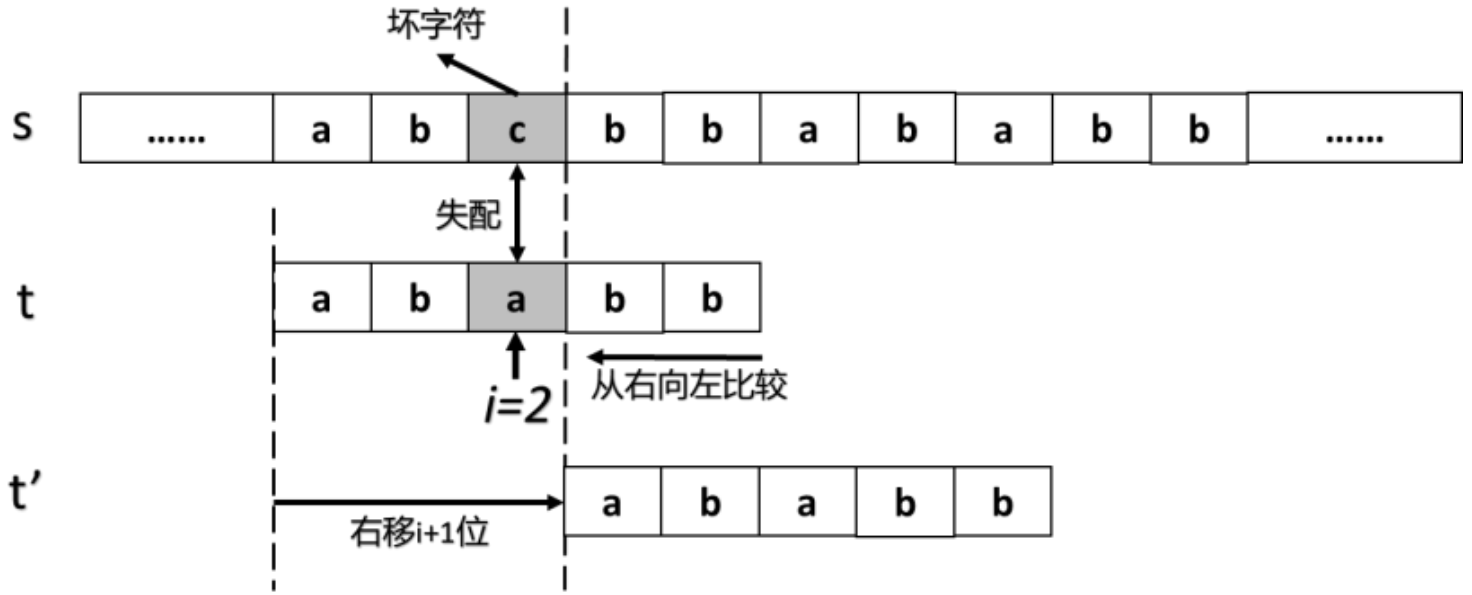
算法思想：先将目标串s和模式串t左对齐，接着从右向左逐位比较。若失配，则按照“坏字符规则”或“好后缀规则”将模式串t向右移动一定位数，再重新从模式串t的最后一位从右向左进行比较，继续以上步骤，直到找到模式串t在目标串s中匹配的位置或未找到匹配情况。



BM算法*

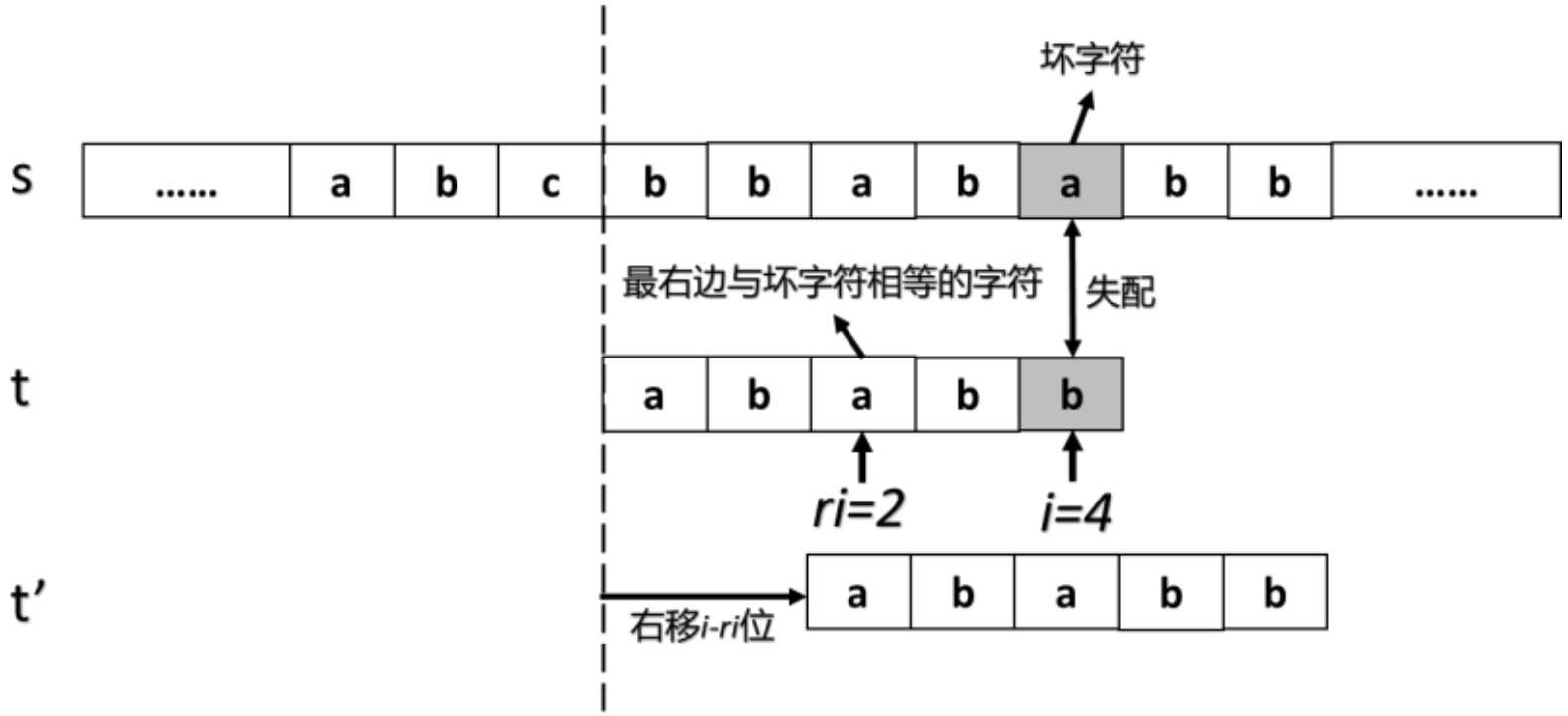
坏字符规则：在模式串与目标串进行匹配的过程中，模式串从右往左的第1个字符、第2个字符已经匹配成功，在匹配从右往左第3个字符时出现失配的情况，此时称在失配位置处目标串中的字符为坏字符，即字符 ‘c’ 为坏字符。分为两种情况：

（1）如果模式串t中并没有出现过这个坏字符。那么显然可以将模式串整体右移至目标串s的坏字符'c'之后的位置，再重新进行匹配。



BM算法*

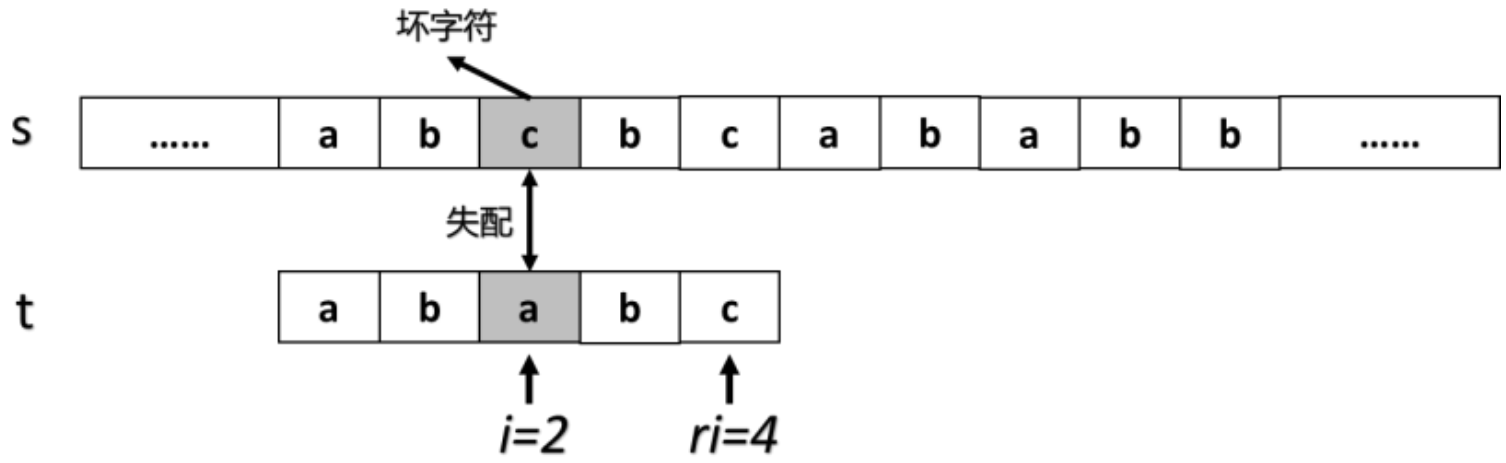
（2）如果模式串t中出现过这个坏字符。如图为上一步移动之后的情况。此时从模式串t的最右边一位开始进行匹配，出现失配情况。此时的坏字符为"a"，并且模式串t中存在与坏字符相同的字符。那么就可以将模式串进行右移，使该字符与坏字符对齐，而后再进行匹配。在使用该策略时，需要选择在模式串中最靠右的与坏字符相同的字符，使其与目标串中的坏字符对齐，这样就保证不会出现移动过多而漏掉匹配的情况。



BM算法*

还可能出现的问题?

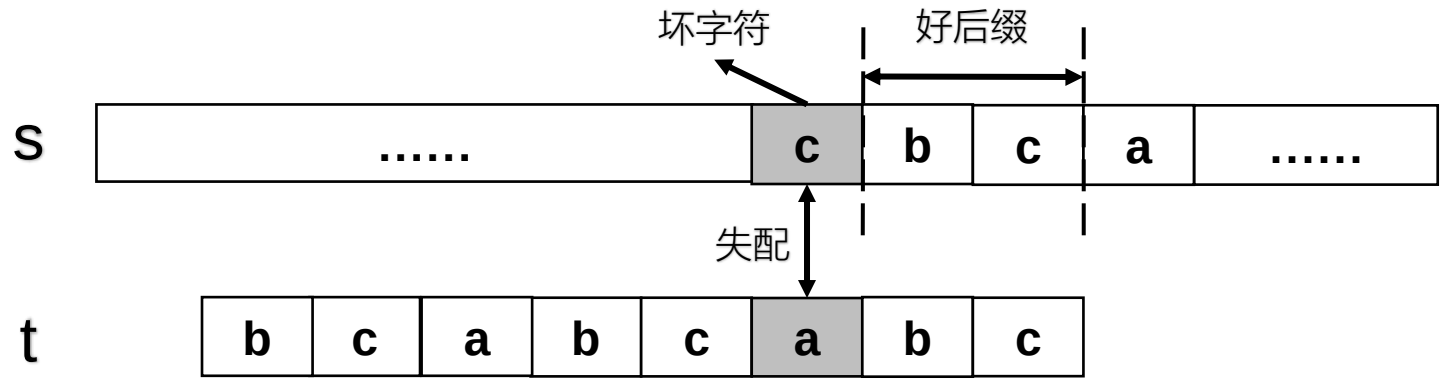
最右边与坏字符相等的字符位置出现在失配位置右边的情况



解决方法：引入“好后缀规则”

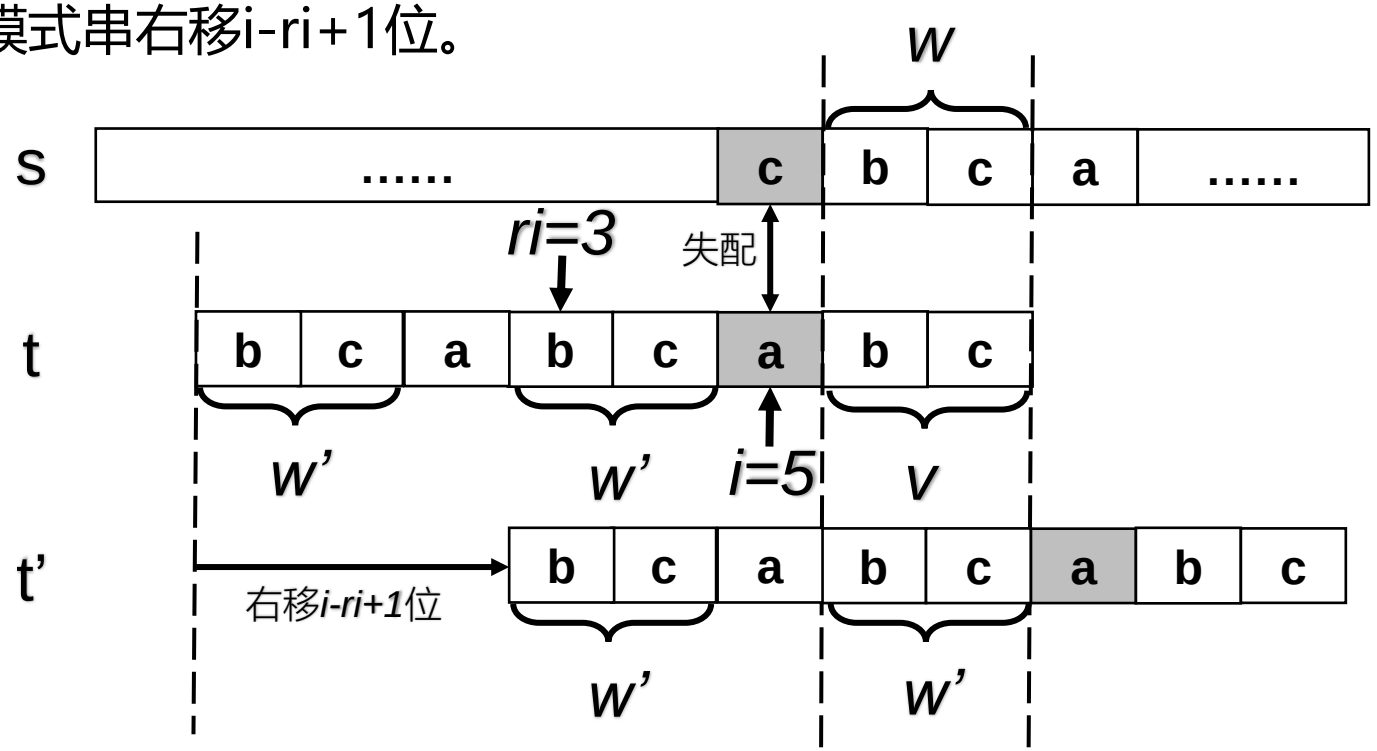
BM算法*

好后缀规则：当模式串t从右向左与目标串s进行匹配时，在从右向左第3个字符处出现失配情况，那么，模式串从右向左的第1个字符、第2个字符已经成功与目标串中的子串匹配，称目标串中已完成匹配的子串为“好后缀”



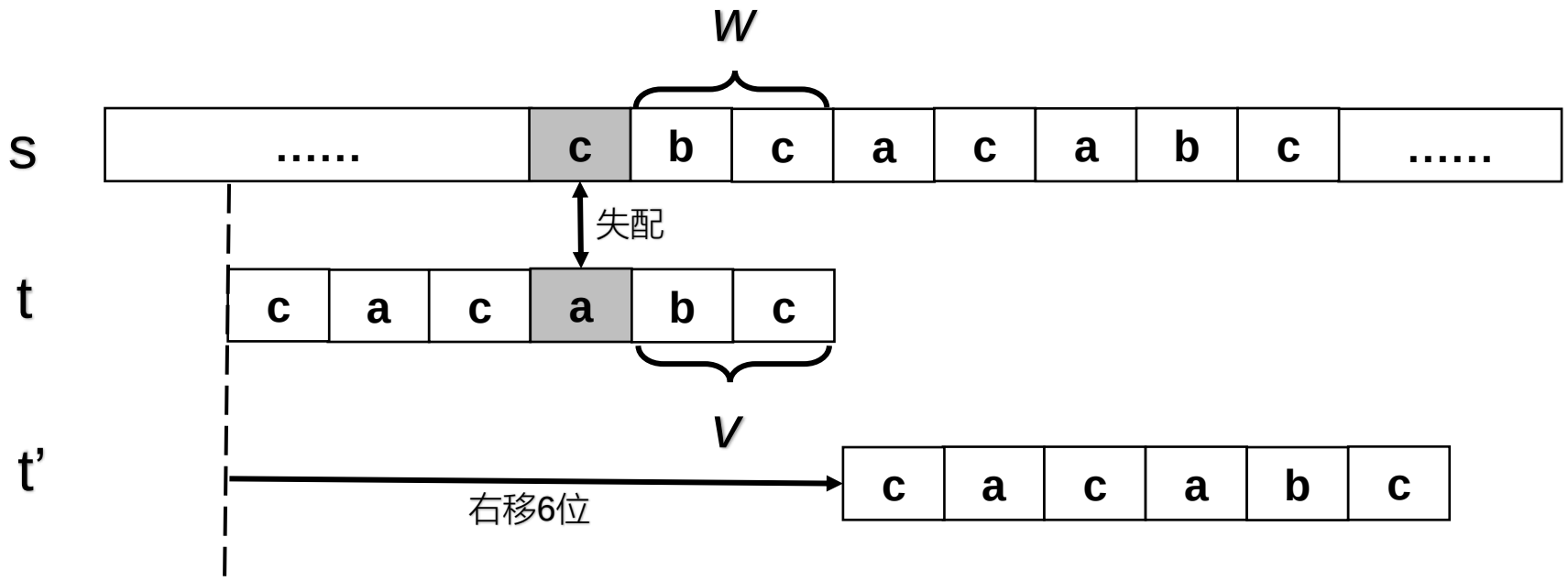
BM算法*

记模式串中失配位置右边的已匹配的字符构成的子串为v，目标串中的好后缀为w，将其作为一个整体，与“坏字符规则”思路一致，在模式串t中寻找不同于v且与w相同的子串w’，将模式串t向右滑动，使模式串中的w’与目标串中的w对齐。如果模式串中存在符合条件的多个子串w’，则为了避免过度移动，选择最右边的w’与目标串对齐，记失配位置为i，最右边的w’的起始位置为ri，则需要将模式串右移i-ri+1位。



BM算法*

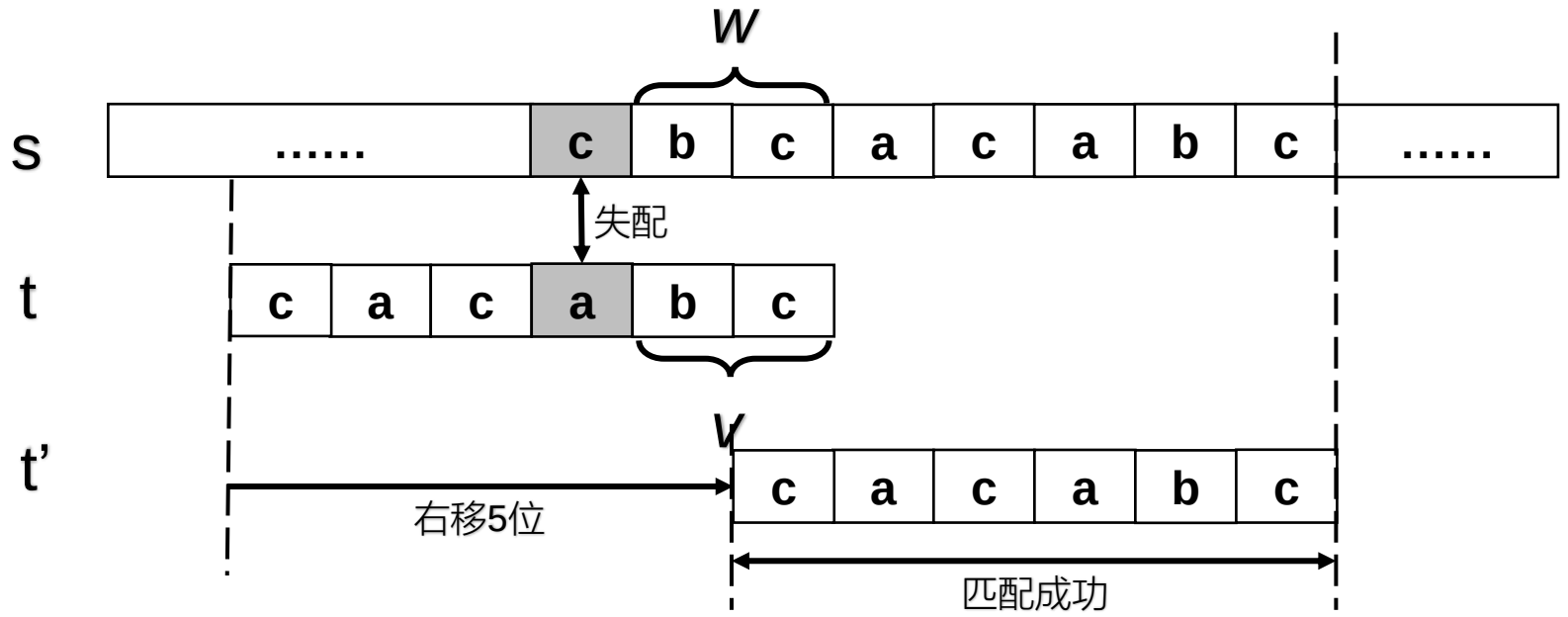
将模式串整体移动到好后缀之后的情况



注意：将模式串整体移动到好后缀之后可能会出现漏掉匹配情况的问题！

BM算法*

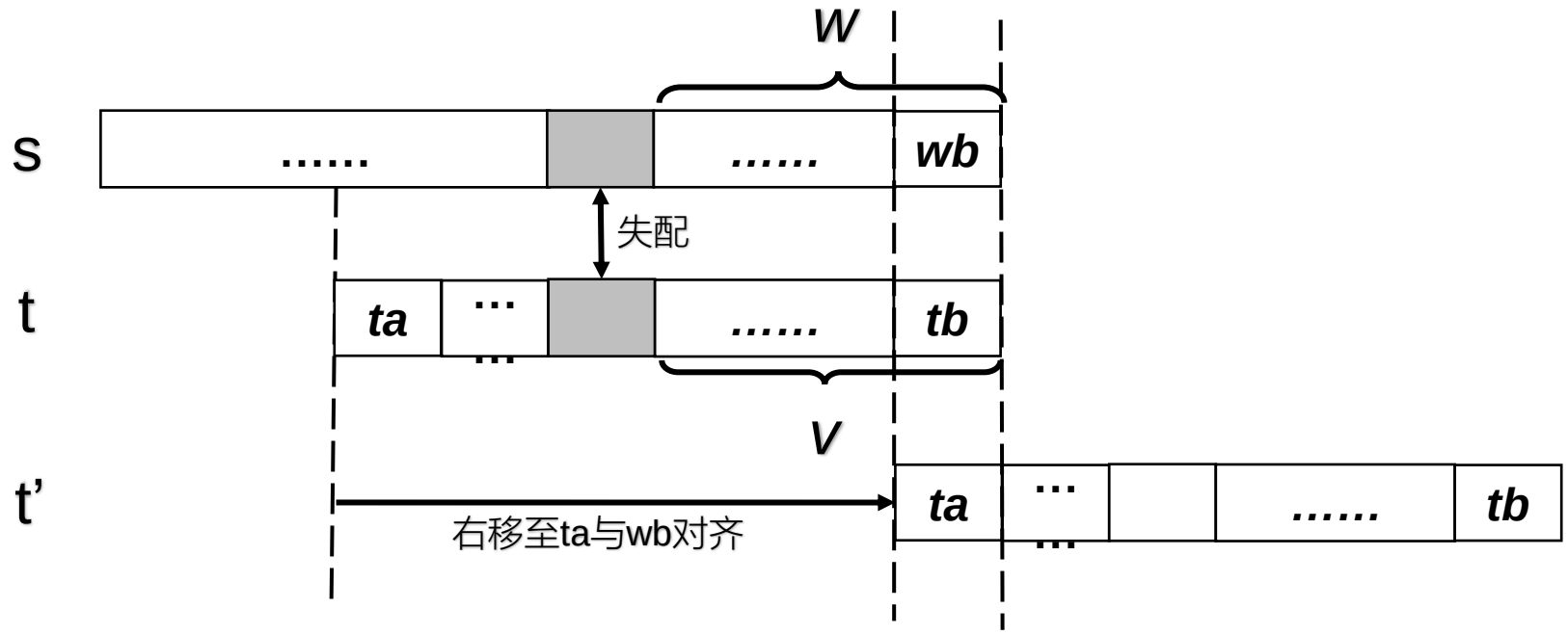
不整体右移到好后缀之后出现匹配成功的情况：将模式串向右移动5位，能够找到匹配情况，说明上文中的右移策略存在过度移动的问题。



通过仔细观察可以发现，这种情况出现的原因是由于模式串t的某个前缀与好后缀w的后缀相同的情况。

BM算法*

解决方法：将模式串t右移，使其前缀与目标串好后缀的后缀相同的部分对齐，其中wb为好后缀w的后缀，ta和tb为模式串t的公共前后缀，ta为模式串t的前缀，tb为模式串t的后缀，wb和tb相同，当模式串中未找到除v之外与w相同的子串时，将模式串t右移，使ta和wb对齐。如果不存在模式串的某前缀与好后缀的某后缀相同的情况，则可直接将t整体右移到好后缀之后。



BM算法*

坏字符规则实现：

坏字符规则中的关键在于获取模式串t最右边与坏字符相等的字符的位置，朴素的处理方法复杂度过高，可考虑对模式串进行预处理，使用一个数组（right数组）来记录每个字符在模式串最右边的位置，通过这种方式，遍历一次模式串，使用O(n)的时间即可完成。

散列表思想：考虑字符串中只包含常见字符的情况，每个字符长度为1字节，所有字符都在ASCII码表的表示范围内。那么可以将right数组长度设置为256(ASCII码的范围为0~255，表示256个不同的常见字符)，用字符的ASCII码作为数组下标，记录每个字符在模式串中最右边的位置。

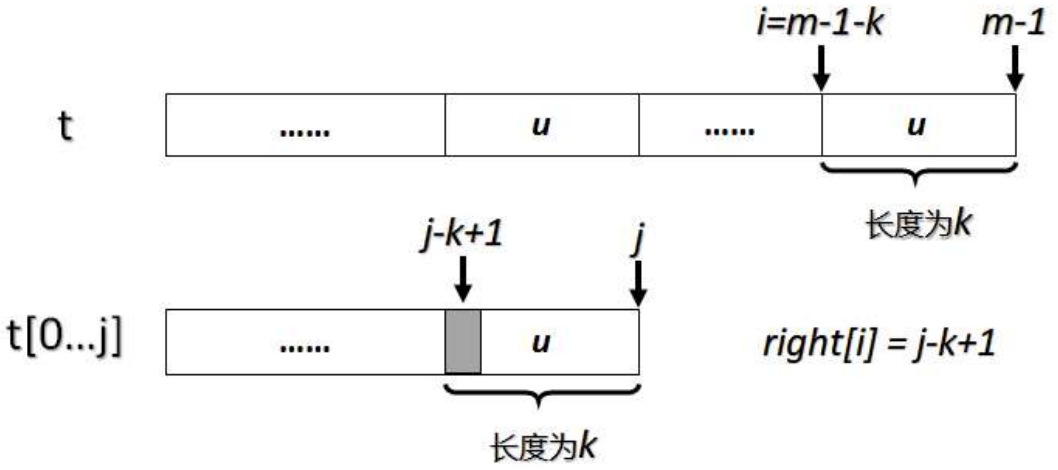
BM算法*

好后缀规则实现：

①在模式串中查找与好后缀匹配的另一个子串

②在好后缀的后缀集合中，找到最长的与模式串的某前缀匹配的后缀。

可先对模式串的所有后缀进行预处理，对每个后缀，从右向左查找第一次出现与后缀匹配的子串，并使用数组记录该子串的起始位置，记该数组为好后缀规则下的right数组。



可以通过已求得的right数组找到好后缀中最长的与模式串的某前缀匹配的后缀

BM算法*

假设在某个位置失配时，基于“坏字符规则”需要将模式串右移a位，基于“好后缀规则”需要将模式串右移b位，那么为了更快完成模式匹配，避免“坏字符规则”可能会出现的向右移动负数位的情况，在BM算法中，选择a和b中较大的作为右移位数。

时间复杂度分析：

预处理阶段：目标串长度为n，模式串长度为m。求坏字符规则和好后缀规则right数组的复杂度均为O(m)。所以BM算法中对模式串进行预处理的复杂度为O(m)。

匹配阶段：最坏情况复杂度为O(n)。

总体上BM算法的最坏情况复杂度为**O(n+m)**。

KR算法*

KR（Karp-Rabin）算法是在朴素的模式匹配算法BF算法基础上引入散列值比较进行加速，避免对字符串的重复比较

算法思想：将目标串s的每个长度为m的子串的散列值与模式串t的散列值比较，如果散列值相同再将子串与模式串逐位比对。

时间复杂度：目标串s长度为n，模式串t长度为m。选择恰当的散列函数，**最好情况下为O(1)** 。KR算法**最坏情况下为O(nm)**，但最坏情况极少出现，总体上，KR算法的效率较高。

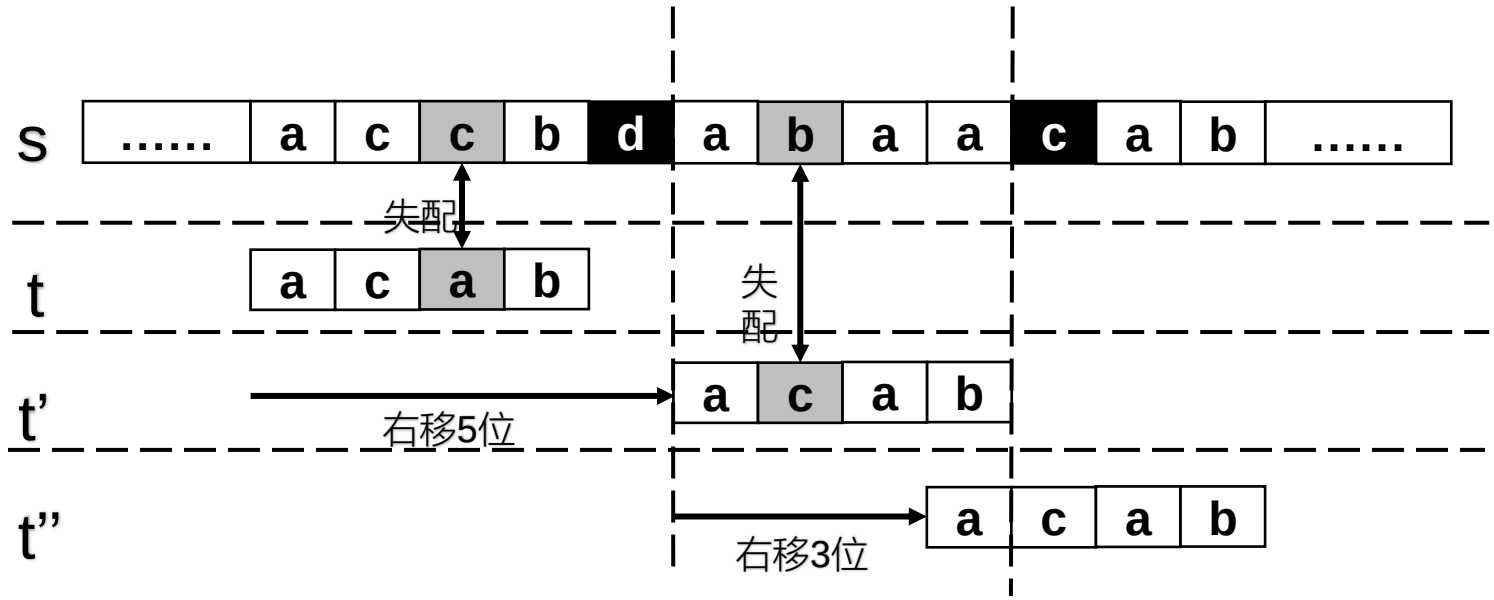
算法4-15 字符串模式匹配的KR算法PatternMatchKR (s,t)

```
输入：目标串s，模式串t
输出：匹配成功返回首个有效位移，匹配失败返回NIL
n ← s.length
m ← t.length
ht ←模式串t的散列值  //这里不对字符串散列进行具体实现，只用变量代替
p ← NIL
for i←0 to n-m do
| hs ← 主串s以i为第一个位置、长度为m的散列值
| if hs = ht then
| |   j ← 0
| | while j < m 且 s.data[i+j] = t.data[j] do
| | | j ← j+1
| | end
| | if j = m then
| | | p = i
| | | break
| | end
| end
end
return p
```

Sunday算法*

Sunday算法是Daniel M.Sunday于1990年提出的一种字符串模式匹配算法，与KMP算法相同，也是在BF算法基础上改进的算法。

算法思想：出现失配时，不同于朴素算法那样只把模式串t右移一位，而是根据目标串s此时与模式串t末尾字符对齐位置的下一位字符在模式串t中出现的情况来决定如何将模式串右移。



Sunday算法*

为了能够快速查询某字符在模式串中的出现情况，设字符集中不同字符的数量为kMaxSize，可通过散列表的方法将字符集中所有字符映射为0~kMaxSize-1中的某个值（称散列值），而后将该散列值作为下标，通过数组形式存储模式串需要向右移动的位数，将该数组称为**偏移表，**记为shift：

$$shift[idx] = \begin{cases} m - \max\{i < m | \mathrm{t}[i] = c\}, & \text{如果字符}c\text{出现在模式串}t\text{中} \\ m + 1, & \text{如果字符}c\text{未出现在模式串}t\text{中} \end{cases}$$

其中idx为字符c通过散列函数映射成的整数

Sunday算法*

Sunday算法：对于目标串s和模式串t，长度分别为n和m，从目标串s的第0位开始查询；假定当前查询索引为i，目标串中的待匹配字符串为子串s[i...i+m]，每次匹配从目标字符串中提取待匹配字符串与模式串进行匹配；若匹配，则返回当前查询索引i；若不匹配，则查看待匹配字符串的后一位字符c，将其通过合适的散列函数映射成一个整数idx；若c存在于模式串t中，则i=i+偏移表[idx]，否则i=i+m。循环以上操作，直到i+m>n，即匹配到目标串的最末尾为止。

时间复杂度：最坏情况时间复杂度为O(nm)。但其在随机数据下表现良好，也是一种较为常用的模式匹配方法。

```
算法4-16 字符串模式匹配的Sunday算法PatternMatchSunday (s,t)
输入：目标串s，模式串t
输出：匹配成功返回首个有效位移，匹配失败返回NIL
注意：这里kMaxSize为常数，是字符集中不同字符的数量
n ← s.length
m ← t.length
p ← NIL
for i←0 to kMaxSize-1 do
| shift[i] ← m+1 //初始化偏移量为最大值
end
for i←0 to m-1 do
| idx ← MapChar(t.data[i]) //用合适的散列函数将t中字符映射成
0~kMaxSize-1中整数
| shift[idx] ← m-i //即t.data[i]的偏移量
end
i ← 0
while i ≤ n-m do
| j ← 0
| while j<m且s.data[i+j]=t.data[j] do
| | j ← j+1
| end
| if j=m then //匹配成功
| | p ← i
| | break
| else
| | idx ← MapChar(s.data[i+m])
| | i ← i+shift[idx]
| end
end
return p
```

拓展延伸*

正则表达式

概念：正则表达式是对字符串的一种形式化描述方式，它通过组合一些具有特殊含义的字符，来实现对特定模式的字符串的描述。可以用来检索满足一定格式的字符串。

结构：正则表达式由普通字符和元字符组成，普通字符包括了英文字母和数字等非元字符的字符，而元字符则具有特殊的含义。

例：元字符“*”表示匹配前面的表达式任意次，“ab*”可匹配“a”，“ab”，“abb”等

实现：对正则表达式建立有穷自动机，让此自动机仅接收满足规则的字符串，再让需要匹配的字符串在自动机上运行即可。

带有通配符的字符串匹配

概念：通配符是一种特殊语句，主要有星号(*)和问号(?)，用来模糊搜索。带有通配符的字符串是一种带有星号和问号的字符串，主要用于字符串的模糊匹配。其中，'*'可以匹配0个或多个连续的任意字符，'?'可以匹配1个任意字符。

例：模式串t为"a*b?c"，字符串"axyzbdc"、"abvc"均能够匹配模式串t。

注意：带有通配符的字符串匹配定义与4.4节中提到的字符串模式匹配不同，其定义为判断目标串s和模式串t是否可以完全匹配，而非子串匹配。

例如，模式串t可以和目标串s的子串匹配，但不能与目标串s完全匹配，则认为模式串t与目标串s失配。

带有通配符的字符串匹配

一种递归算法：

1. s或者t其中一个已经到末尾了，那么如果t的剩余字符都是'*'，返回匹配，否则返回不匹配；

2. 如果s的当前字符和t的当前字符相等，继续向后移动；

3. 当s的当前字符和t的当前字符不相等，分为三种情况：

(1) t的当前字符不是'*'或'?'，返回不匹配；

(2) 若t的当前字符是'?'，继续向后移动；

(3) t的当前字符是'*'，那么可跳过s的0到多个字符，再递归判断是否匹配

时间复杂度：由于对每次递归判断过程，目标串s和模式串t的指针i和j都是单调递增的，所以复杂度为O(n+m)。由于最多递归O(n)次，所以该算法的时间复杂度为**O(n(n+m))**。

应用场景：基因测序

示例场景：生物信息学中已知一串复杂的病毒RNA序列，其规模可以达到百万级别。致病性的RNA序列段规模达到万级，此时科学家需要判断这串病毒的RNA序列中是否包含致病序列段。

解决方法：将复杂的病毒RNA序列认为是目标串s，致病性的RNA序列段认为是模式串t，使用KMP算法解决模式匹配问题，判断模式串t是否为目标串s的子串，如果是，则可以认定该病毒具有致病性，反之则认为该病毒没有致病性。

小结

各种匹配算法的比较：

- 朴素算法通过逐位比较的方式进行匹配，思路最容易理解，算法时间复杂度为O(nm)；
- KMP算法利用利用字符串特征向量加速匹配过程，在任何场景下算法复杂度能够达到稳定的O(n+m),在小字符集模式应用场景下表现尤其突出；
- BM算法基于坏字符与好后缀规则来优化匹配算法，最坏情况复杂度为O(n+m)，更适用于大字符集模式、字符较为随机的应用场景；
- KR算法和Sunday算法分别利用散列表技术和类似于坏字符规则的方法来提高匹配效率，虽然最坏情况下时间复杂度均为O(nm)，但在字符随机出现的情况下表现良好，并且由于其简单明了、实现简单的优势，在工程实践中得到广泛关注。