假设字符串s长度为n, 字符串ms长度为m, 要在s中查找ms

1. BF算法，即暴力求解

需要n-m+1次， ms在s中不断的右移匹配。

尽管理论上，BF 算法的时间复杂度很高，是 O(n\*m)，但在实际的开发中，它却是一个比较常用的字符串匹配算法。

为什么这么说呢？原因有两点。

第一，实际的软件开发中，大部分情况下，模式串和主串的长度都不会太长。

而且每次模式串与主串中的子串匹配的时候，当中途遇到不能匹配的字符的时候，就可以就停止了，

不需要把 m 个字符都比对一下。所以，尽管理论上的最坏情况时间复杂度是 O(n\*m)，但是，统计意义上，

大部分情况下，算法执行效率要比这个高很多。

第二，朴素字符串匹配算法思想简单，代码实现也非常简单。简单意味着不容易出错，如果有 bug 也容易暴露和修复。

在工程中，在满足性能要求的前提下，简单是首选。这也是我们常说的KISS（Keep it Simple and Stupid）设计原则。

2. RK算法

RK 算法的思路是这样的：我们通过哈希算法对主串中的 n-m+1 个子串分别求哈希值，然后逐个与模式串的哈希值比较大小。

因为hash值只需比较数值大小，因此速度更快。但计算了hash需要保存，会增加空间复杂度，典型的用空间换时间。

另外计算hash也会带来一定的复杂度。

为了提升效率，这就需要哈希算法设计的非常有技巧了。

我们假设要匹配的字符串的字符集中只包含 K 个字符，我们可以用一个 K 进制数来表示一个子串，

这个 K 进制数转化成十进制数，作为子串的哈希值。表述起来有点抽象，我举了一个例子，看完你应该就能懂了。

比如10进制 "369" = 3 \* 100 + 6 \* 10 + 9

a-z就可以 "abc" = a \* 16 \* 16 + b \* 16 + c

这种哈希算法有一个特点，在主串中，相邻两个子串的哈希值的计算公式有一定关系。

也就是说，我们可以使用 s[i:j]的哈希值很快的计算出 s[i+1:j+1]的哈希值。

3. BM算法

它是一种非常高效的字符串匹配算法，有实验统计，它的性能是著名的KMP 算法的 3 到 4 倍。

BM 算法的原理很复杂，比较难懂。

a. 坏字符规则

1) 从模式串的末尾往前倒着匹配，当发现某个字符没法匹配的时候，我们把这个没有匹配的字符叫作坏字符（主串中的字符）。

比如主串s = "abcacabdac" 中查找模式串ms = "abd", 第一次进行查找时，"c"就是坏字符

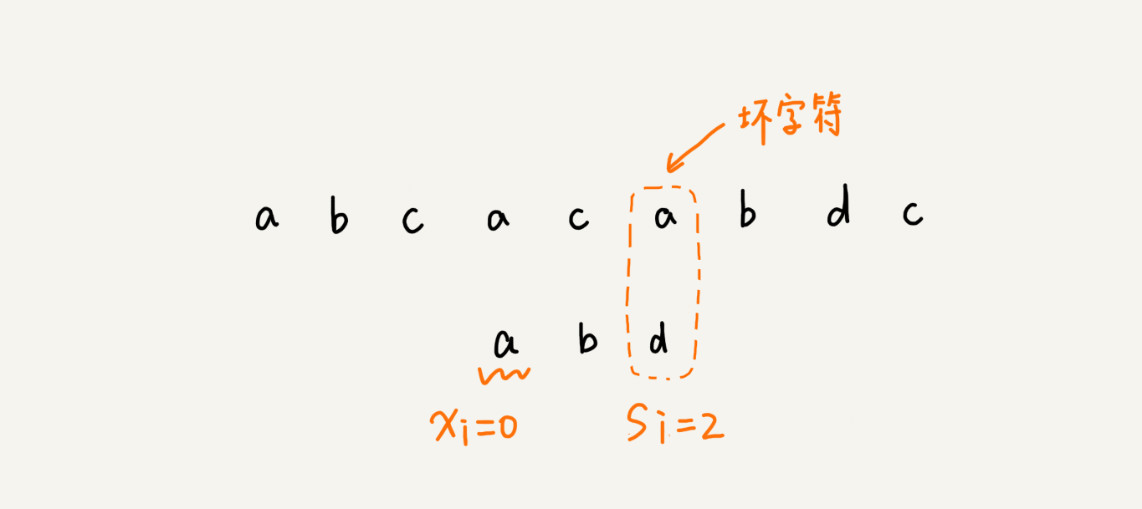
2) 我们拿坏字符 c 在模式串中查找，发现模式串ms中并不存在这个字符，也就是说，

字符 c 与模式串中的任何字符都不可能匹配。这个时候，我们可以将模式串直接往后滑动3位，

将模式串滑动到 c 后面的位置，再从模式串的末尾字符开始比较。

3）于是ms = "abd"和s中"aca"子串开始比较，同理可得到坏字符"a"。这时由于"a"在模式字符串ms中可以找到,

就不能后移3位了，而是后移2位，让"abd"和"aca"中的第二个a对齐。于是就找到了abd。



总结起来：

**当发生不匹配的时候，我们把坏字符对应的模式串中的字符下标记作 si。**

**如果坏字符在模式串中存在，我们把这个坏字符在模式串中的下标记作 xi；如果不存在，我们把 xi 记作 -1。**

**那模式串往后移动的位数就等于 si-xi。**

（注意，我这里说的下标，都是字符在模式串的下标）。

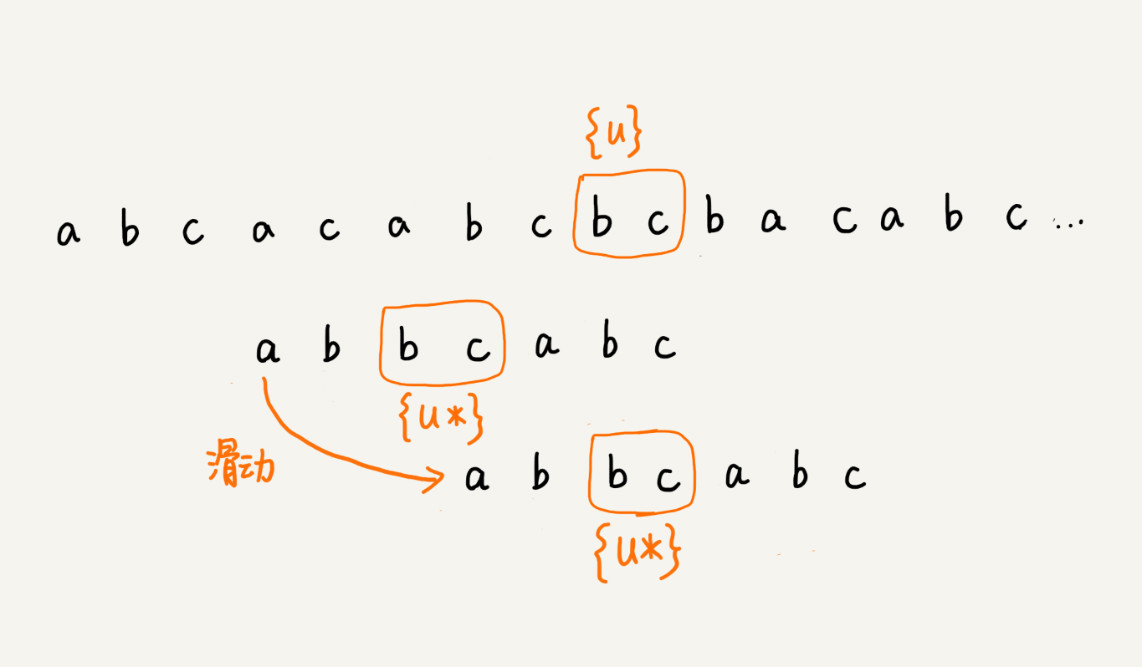
如果坏字符在模式串里多处出现，那我们在计算 xi 的时候，**选择最靠后的那个，因为这样不会让模式串滑动过多**，导致本来可能匹配的情况被滑动略过。

b. 好后缀规则

单纯使用坏字符规则还是不够的。因为根据 si-xi 计算出来的移动位数，有可能是负数，

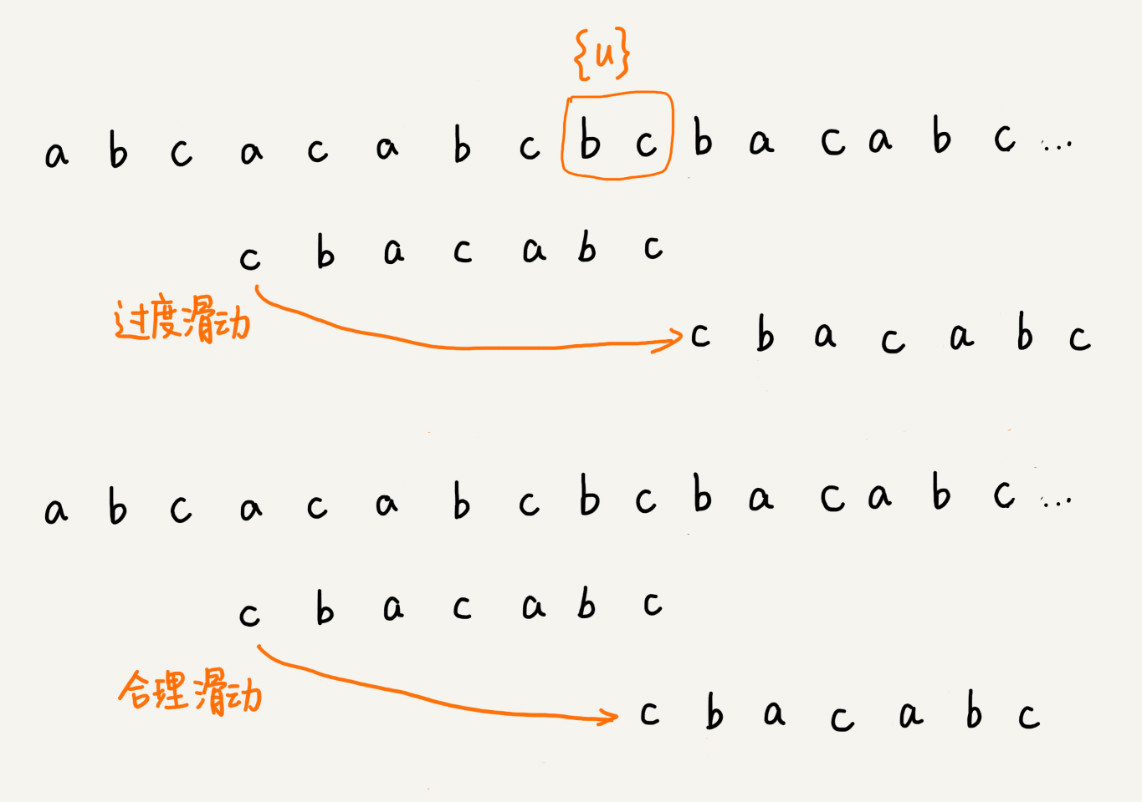
比如主串是 aaaaaaaaaaaaaaaa，模式串是 baaa。不但不会向后滑动模式串，还有可能倒退。

所以，BM 算法还需要用到“好后缀规则”。



我们把已经匹配的 bc 叫作好后缀，记作{u}。我们拿它在模式串中查找，如果找到了另一个跟{u}相匹配的子串{u\*}，那我们就将模式串滑动到子串{u\*}与主串中{u}对齐的位置。

如果在模式串中找不到另一个等于{u}的子串，我们不仅要看好后缀在模式串中，是否有另一个匹配的子串，我们还要考察好后缀的后缀子串，是否存在跟模式串的前缀子串匹配的。



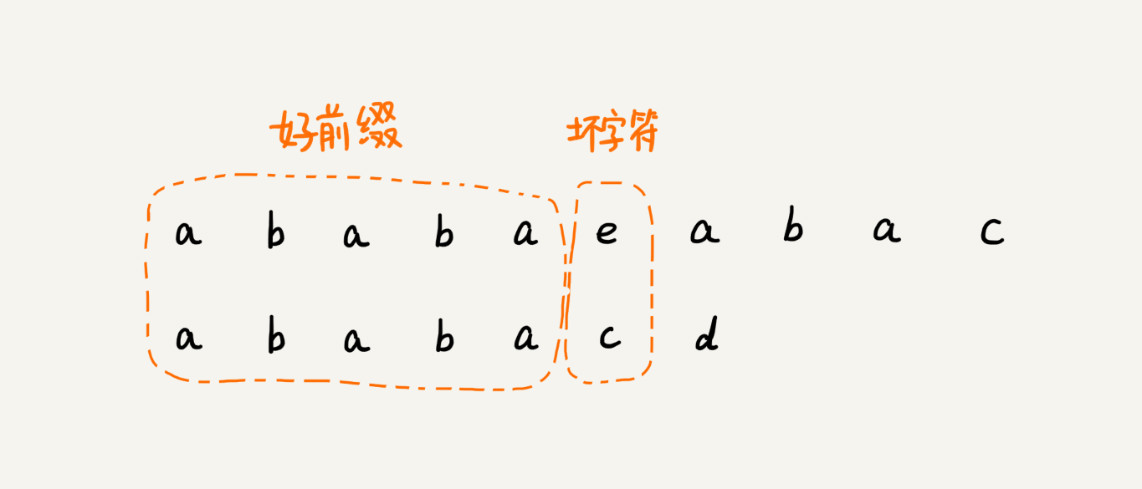
所谓某个字符串 s 的后缀子串，就是最后一个字符跟 s 对齐的子串，比如 abc 的后缀子串就包括 c, bc。所谓前缀子串，就是起始字符跟 s 对齐的子串，比如 abc 的前缀子串有 a，ab。**我们从好后缀的后缀子串中，找一个最长的并且能跟模式串的前缀子串匹配的，假设是{v}，然后将模式串滑动到如图所示的位置**。

1. 总结

我们可以**分别计算好后缀和坏字符往后滑动的位数，然后取两个数中最大的**，作为模式串往后滑动的位数。这种处理方法还可以避免我们前面提到的，根据坏字符规则，计算得到的往后滑动的位数，有可能是负数的情况。这就是BM算法了。

1. KMP算法

在模式串和主串匹配的过程中，把不能匹配的那个字符仍然叫作坏字符，把已经匹配的那段字符串叫作好前缀。



我们只需要拿好前缀本身，在它的后缀子串中，查找最长的那个可以跟好前缀的前缀子串匹配的。

