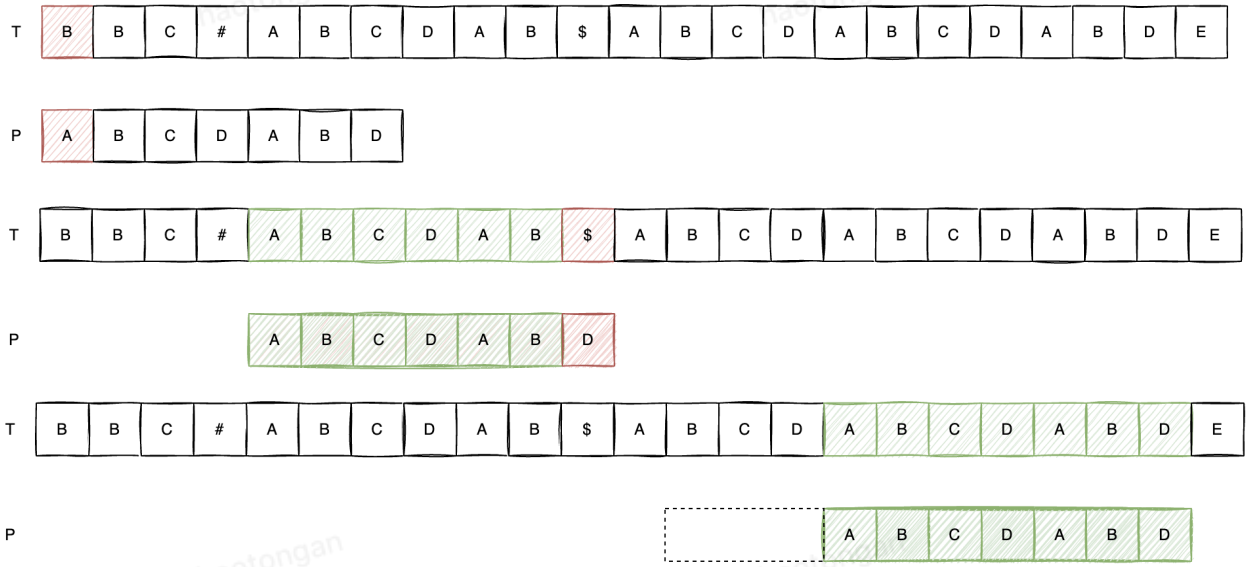
String Search

一、简介

1.1 Background

字符串匹配在文本处理的广泛领域中是一个非常重要的主题。字符串匹配包括在文本中找到一个，或者更一般地说，所有字符串(通常来讲称其为模式)的出现。该模式表示为p=p[0..m-1];它的长度等于m。文本表示为t=t[0..n-1],它的长度等于n。两个字符串都建立在一个有限的字符集上。

一个比较常见的字符串匹配方法工作原理如下。在一个大小通常等于m的窗口帮助下扫描文本。首先将窗口和文本的左端对齐，然后将窗口的字符与文本中的字符进行比较，这一特定的工作被称为尝试，在完全匹配或不匹配之后，将窗口移到右侧。继续重复同样的过程，直到窗口的右端超过文本的右端，一般称为滑动窗口机制。



1.2 Brute force

BF算法检查文本中0到n-m之间的所有位置，是否有模式从那里开始出现。然后，在每次尝试之后，它将模式串向右移动一个位置。

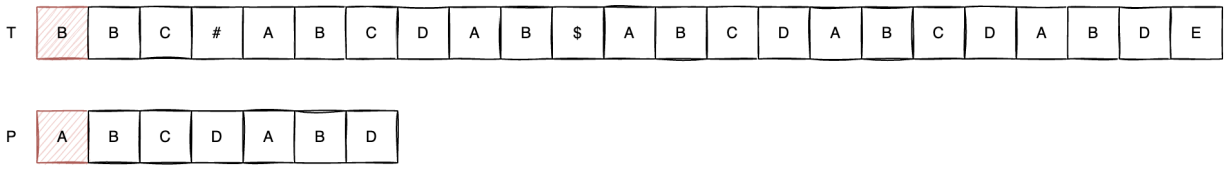
BF算法不需要预处理阶段，除了模式和文本之外，还需要一个恒定的额外空间。在搜索阶段，文本字符比较可以以任何顺序进行。该搜索阶段的时间复杂度为O(mn)。

Brute force

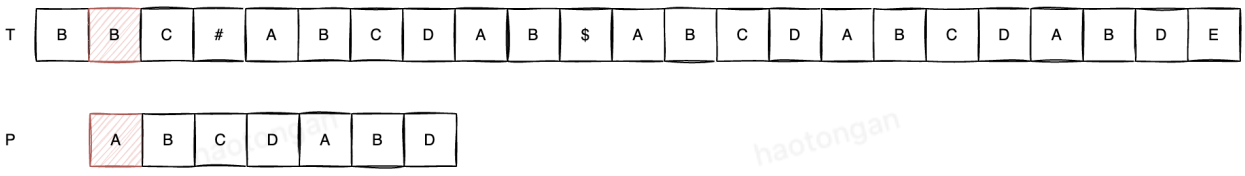
```java
public static int strMatch(String s, String p){
    int i = 0, j = 0;
    while(i < s.length() && j < p.length()){
        if(s.charAt(i) == p.charAt(j)){
            i++;
            j++;
        }else{
            i = i - j + 1;
            j = 0;
        }
        if (j == p.length()){
            return i - j;
        }
    }
    return -1;
}
```

二、KMP

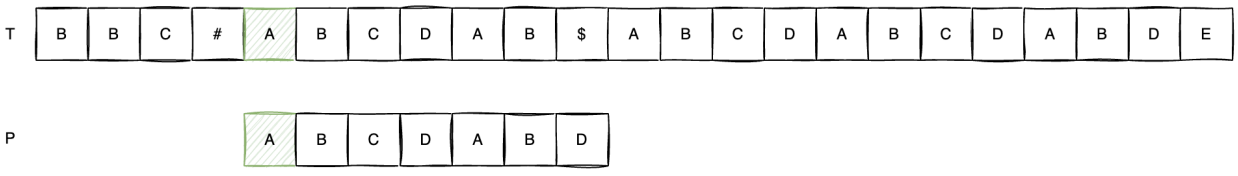
先回顾下brute force中匹配的情况。我们在文本串BBC#ABCDAB$ABCDABCDABDE中查找模式串ABCDABD，文本串中第1个字符“B”与模式串中第1个字符“A”不匹配，所以我们将模式传后移一位。



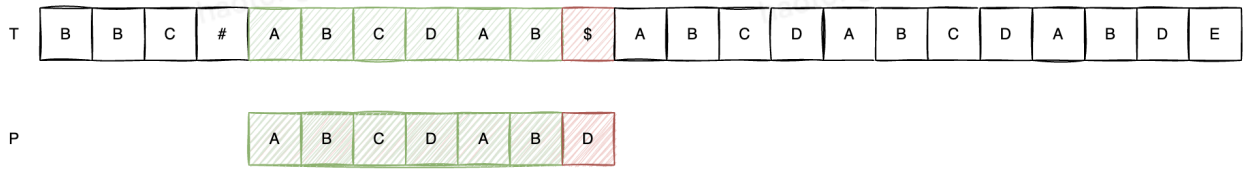
文本串中的第2个字符“B”和模式串中的第一个字符“A”不匹配，继续后移。



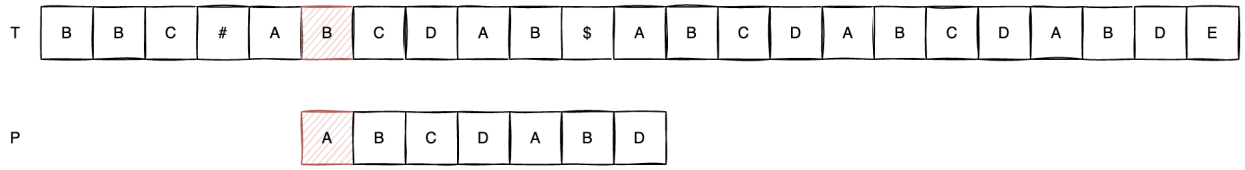
基于这种方式不断比较并且移动，我们发现文本串中的第5个字符“A”和模式串中的第1个字符“A”是匹配的，那么继续比较文本串和模式串的下一个字符。



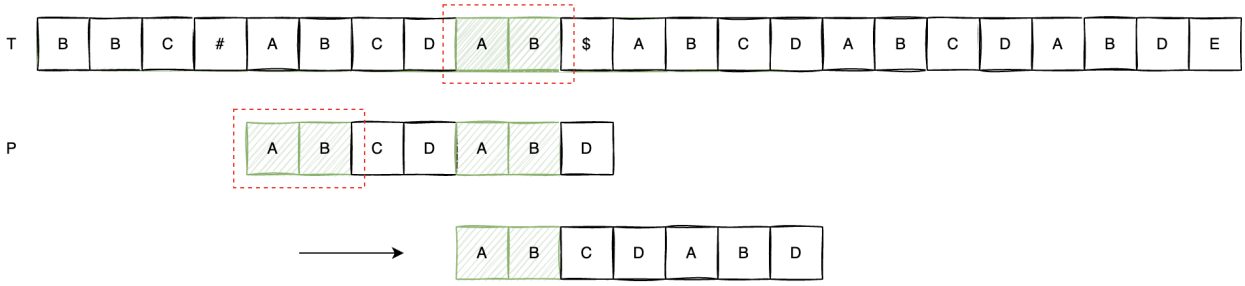
不断比较之后我们发现，文本串中的字符“$”和模式串中的最后一个字符“D”不匹配。



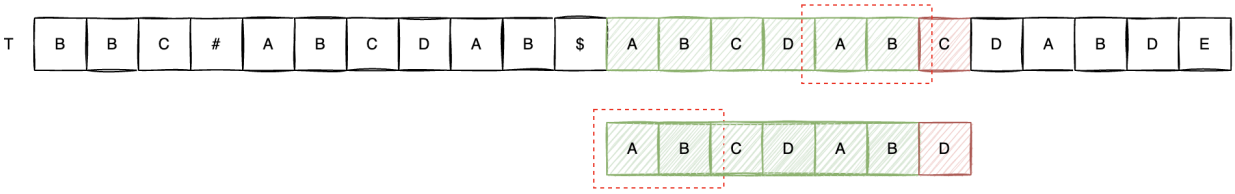
根据BF算法，我们应该继续将模式串向后移动一位，然后从头开始重新比较。



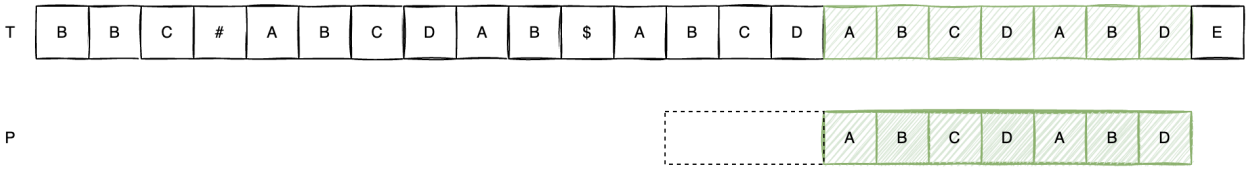
那我们不妨观察下，上次匹配失败的情况，当文本串中“$”与模式串中“D”不匹配时，我们其实已经完成了6次匹配，也就是说我们在文本串和模式串中已经找到了"ABCDAB"。同时我们可以发现模式串中前缀“AB”是可以和文本串中已匹配成功部分的后缀“AB”相匹配，我们利用这个信息，可以把模式串右移多位，而不仅仅是1位来去继续匹配（换句话说，我们不需要回退文本串的搜索位置），这加快了搜索速率。



同样的，当搜索到下面情况时，文本串中的字符“C”和模式串中的字符“D”不匹配，利用已知的信息，我们右移模式串，不回退搜索位置，继续去查找匹配。



最终，查找成功。



简单来说，文本串和模式串匹配失败时，kmp算法并没有像bf算法描述中一样，将模式串右移1位，从头重新进行搜索，而是利用已匹配信息，不回退文本串的搜索位置，继续将模式串向后移动，减少比较次数，提高了效率。那么当匹配失败时，模式串究竟要向后移动多少位呢?

2.1 前缀函数

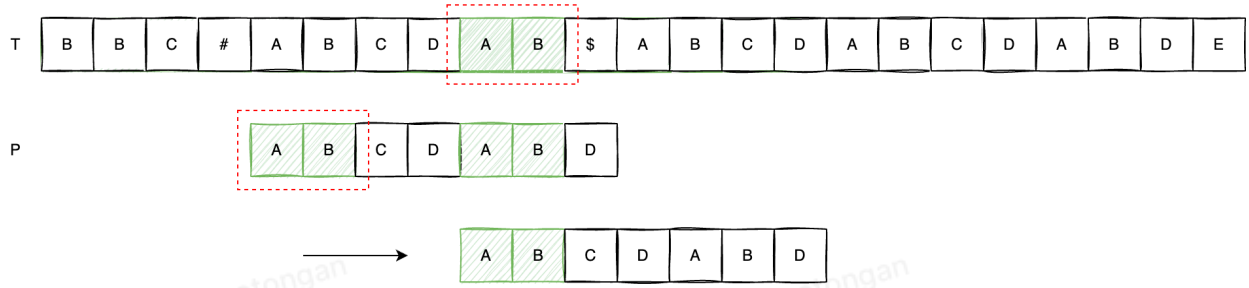
前缀是指从串首开始到某个位置结束的一个特殊子串。字符串S以i结尾的前缀表示为Prefix(S,i)，也就是Prefix(S,i)=S[0..i]。

真前缀指除了S本身的S的前缀。

后缀是指从某个位置开始到整个串末尾结束的一个特殊子串。字符串S的从i开头的后缀表示为Suffix(S,i)，也就是Suffix(S,i)=S[i..|S|-1]。

真后缀指除了S本身的S的后缀。

回到上文kmp算法匹配流程中，当文本串和模式串匹配失败时，我们右移模式串的位数是多少呢？或者说，当文本串中字符与模式串中字符匹配失败时，应该重新跟模式串中哪个字符再进行匹配呢？



上面的例子文本串中$与模式串中D匹配失败，而由于已经匹配成功了“ABCDAB”这6个字符，我们发现可以将模式串右移4位再进行比较，或者说此时，当匹配至模式串第7个字符失败后，可以重新和模式串的第3个字符，也就是“C”进行比较，这是由于文本串中的“AB”恰好和模式串中的前缀“AB”相匹配。而且我们发现匹配失败前文本串中的“AB”和已匹配的模式串中的后缀“AB”也是相匹配的。所以实际上我们根据模式串自身的特点，就能知道匹配失败时如何去匹配新的位置。

我们定义数组prefix，其中prefix[i]表示以S.charAt(i)为结尾的即S[0..i]中最长的相同真前后缀的长度。以字符串“aabaaab”为例：

i=0时，子串“a”无真前后缀，prefix[0]=0

i=1时，子串“aa”，其中[a]a和a[a]最长的相同真前后缀为a，prefix[1]=1

i=2时，子串“aab”无相同的真前后缀，prefix[2]=0

i=3时，子串“aaba”，其中[a]aba aab[a]最长的相同真前后缀为a，prefix[3]=1

i=4时，子串“aabaa”，其中 [aa]baa aab[aa] 最长的相同真前后缀为aa，prefix[4]=2

i=5时，子串“aabaaa”，其中[aa]baaa aaba[aa] 最长的相同真前后缀为aa，prefix[5]=2

i=6时，子串“aabaaab”，其中[aab]aaab aaba[aab]最长的相同真前后缀为aab，prefix[6]=3

i	0	1	2	3	4	5	6
s[i]	a	a	b	a	a	a	b
prefix[i]	0	1	0	1	2	2	3

上文匹配的prefix数组如下：

i	0	1	2	3	4	5	6
s[i]	A	B	C	D	A	B	D
prefix[i]	0	0	0	0	1	2	0

如何求解prefix呢，很容易想到一种方法是，我们使用两个for循环来遍历给定字符串的前缀中的真前缀和真后缀，内部去比较真前缀和真后缀是否相同。即便我们从最长的真前后缀来尝试匹配，这个方法的时间复杂度还是很高。

```
1   public static int[] getPrefix(String str){
2           int[] res = new int[str.length()];
3           for(int i = 1; i < res.length; ++i){
4               for(int j = i; j > 0; --j){
5                   if (str.substring(0, j).equals(str.substring(i-j+1,i+1))){
6                       res[i] = j;
7                       break;
8                   }
9               }
10          }
11          return res;
12      }
```

2.2 第一个优化

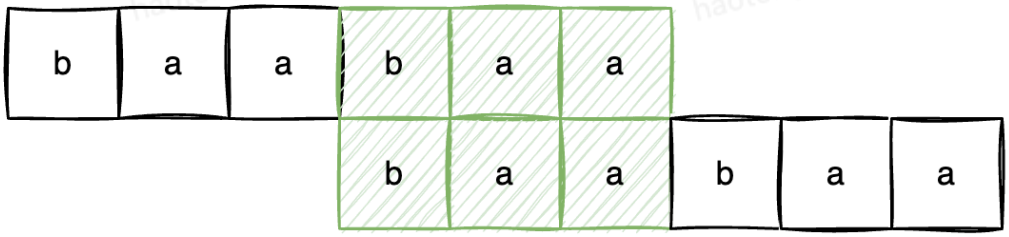
我们观察下由s[i]至s[i+1]求解最长的真前后缀匹配情况变化。

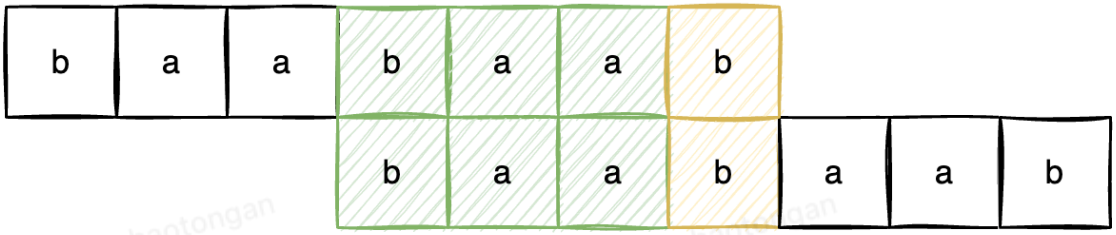
```
// compute "ABCDA" -> compute "ABCDAB"
// A         A       <-"ABCDA"时最长前缀、后缀匹配
// AB        DA
// ABC       CDA
// ABCD      BCDA
// ->
// A         B
// AB        AB      <-"ABCDAB"时最长前缀、后缀匹配
// ABC       DAB
// ABCD      CDAB
// ABCDA     BCDAB

// compute "ABCDA" -> compute "ABCDAP"
// A         A       <-"ABCDA"时最长前缀、后缀匹配
// AB        DA
// ABC       CDA
// ABCD      BCDA
// ->
// A         P
// AB        AP
// ABC       DAP
// ABCD      CDAP
// ABCDA     BCDAP
// 无匹配

// A->AB
// 也就是说最好的情况下，以s[i]为结尾的最长的相同的真前后缀长度，一定是以s[i-1]为结尾的最大的相同的真前后缀相同的长度+1
```

根据上面的描述，在尝试匹配真前后缀的时候，我们可以减少循环次数。

```java
public static int[] getPrefix1(String str){
    int[] prefix = new int[str.length()];
    prefix[0] = 0;
    for (int i = 1; i < str.length(); ++i){
        for(int j = prefix[i-1] + 1; j > 0; --j){
            if (str.substring(0, j).equals(str.substring(i-j+1, i+1))){
                prefix[i] = j;
                break;
            }
        }
    }
    return prefix;
}
```

考虑一种情况，计算字符串“baabaab”的prefix的时候，在计算i=5的时候，我们已经完成了“baa”的比较，当计算i=6的时候，我们比较前缀“baab”和后缀“baab”，但是在上一次比较，我们知道前缀“baa”和后缀“baa”已经匹配了。





为了减少这种重复的匹配，我们考虑一下利用双指针来不断的去比较所指的两个字符

```java
// if(s.charAt(i) == s.charAt(j))
//      prefix[i] = prefix[j-1] + 1;
//      or
//      prefix[i] = j + 1;
// }
```

```java
public static int[] getPrefix2(String str){
    int[] prefix = new int[str.length()];
    int j = 0;
    int i = 1;
    while(i < str.length()){
        if (str.charAt(j) == str.charAt(i)){
            j++;
            prefix[i] = j;
            i++;
        }else{
            // 匹配失败时,
            while(j > 0 && !str.substring(0, j).equals(str.substring(i-j+1,
                j--;
            }
            prefix[i] = j;
            i++;
        }
    }
    return prefix;
}
```

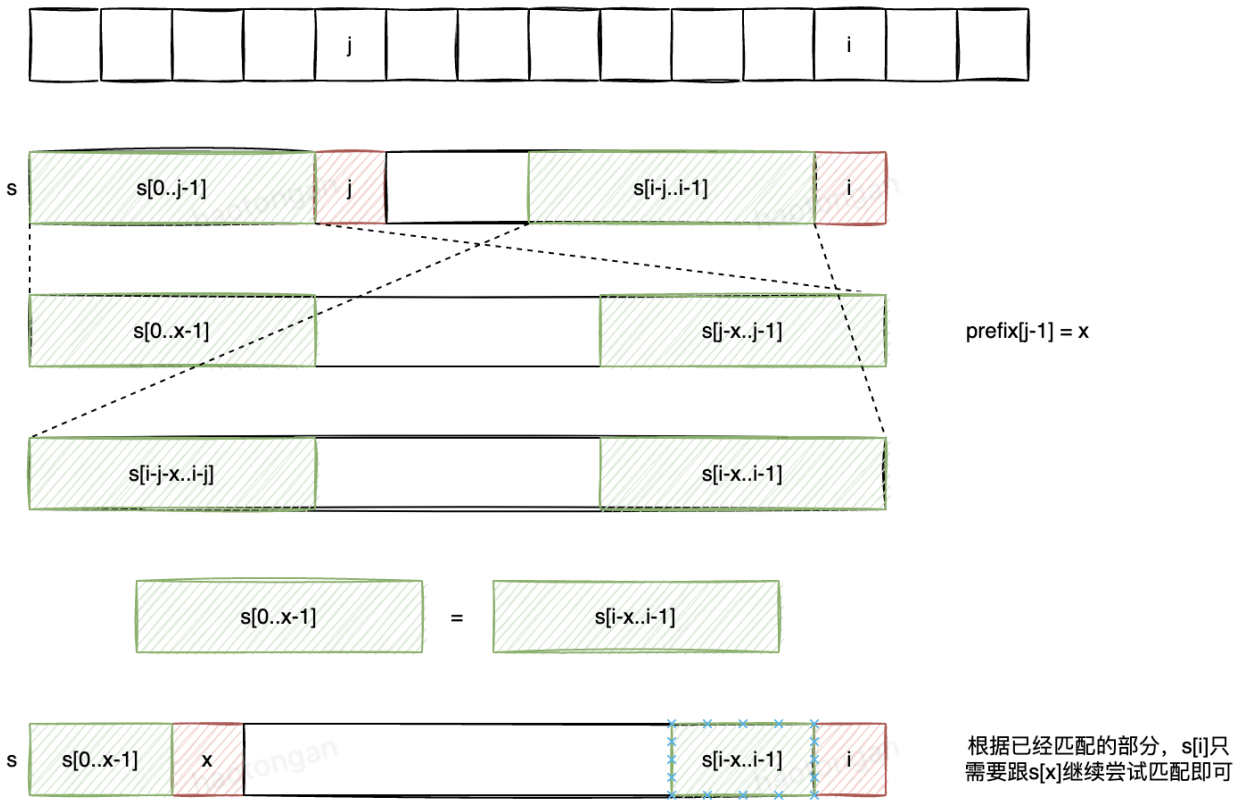
2.3 第二个优化

上面的优化是针对匹配成功时候的情况，那么匹配失败时，难道真的需要重新去枚举其他的真前后缀，来去不断的尝试匹配吗？我们观察下，匹配失败时，能否利用前面已经计算完的结果呢？

当s[j]!=s[i]的时候，我们是知道s[0..j-1]和s[i-j..i-1]是相同的，到这里再回想一下prefix数组的定义，prefix[j-1]表示的是以s.charAt(j-1)字符为结尾的即s[0..j-1]中最长的相同真前后缀的长度，如果prefix[j-1]=x(x!=0)，我们很容易得到s[0..x-1]和s[j-x..j-1]是相同的。

再将s[i-j..i-1]展开来看一下，因为我们知道s[0..j-1]和s[i-j..i-1]是相同的，所以s[i-j..i-1]也同样存在相同的真前后缀，即真前缀s[i-j-x..i-j]以及真后缀s[i-x..i-1]，而且由于s[0..x-1]和s[j-x..j-1]是相同的，s[j-x..j-1]和s[i-x..i-1]是相同的（整体相同，对应的部分也是相同的），可以容易得到s[0..x-1]和s[i-x..i-1]是相同的。

再回到原始的字符串上来观察，s[0..x-1]正是字符串s的真前缀，而s[i-x..i-1]是以i-1为结尾的真后缀，由于这两部分相同，我们更新j=x=prefix[j-1]，准确找到已经匹配的部分，继续完成后续的匹配即可。



代码实现如下

```java
public static int[] getPrefix4(String str){
    int[] prefix = new int[str.length()];
    int j = 0;
    int i = 1;
    while(i < str.length()){
        if (str.charAt(j) == str.charAt(i)){
            // 更新j，同时j++也正是已匹配的最大长度
            j++;
            prefix[i] = j;
            i++;
        }else if(j == 0){
            // 当str.charAt(j) != str.charAt(i) && j == 0时，后移i即可
            i++;
        }else{
            // 找到已匹配的部分，继续匹配即可
            j = prefix[j-1];
        }
    }
    return prefix;
}
```

2.4 求解next

很多kmp算法的讲解都提到了next数组，那么实际上next数组求解和上面的prefix求解本质是一样的，next[i]实际上就是以i-1为结尾的最长的相同真前后缀的长度。

定义next[j]为当s[i] != p[j]时，需要跳转匹配的模式串的索引，特别的当next[0] = -1

```
public static int[] getNext(String str){
    int[] next = new int[str.length()+1];
    int i = 1;
    int j = 0;
    // next[0] = -1 指代匹配失败，更新文本串索引+1
    next[0] = -1;
    while(i < str.length()){
        if (j == -1 || str.charAt(i) == str.charAt(j)){
            i++;
            j++;
            next[i] = j;
        }else{
            j = next[j];
        }
    }
    return next;
}
```

i	0	1	2	3	4	5	6	7
s[i]	A	B	C	D	A	B	D	
prefix[i]	0	0	0	0	1	2	0	
next[i]	-1	0	0	0	0	1	2	0

2.5 完整代码

```java
public static int search(String s, String p){
    int[] next = getNext(p);
    int i = 0, j = 0;
    while(i < s.length() && j < p.length()){
        if (j == -1 || s.charAt(i) == p.charAt(j)){
            i++;
            j++;
        }else{
           j = next[j];
        }
        if (j == p.length()){
            return i - j;
        }
    }
    return -1;
}
```

2.6 优化next

以上面的next数组为例，当i=5，匹配失败时，应该跳转i=1进行比较，但是我们知道s[5]=s[1]="B"，这样匹配下去也是必定会失败的，基于这一点，还可以简单优化下next数组的求解过程。

```java
public static int[] getNext1(String str){
    int[] next = new int[str.length()+1];
    int i = 1;
    int j = 0;
    next[0] = -1;
    while(i < str.length()){
        if (j == -1 || str.charAt(i) == str.charAt(j)){
            i++;
            j++;
            if (i < str.length() && str.charAt(i) != str.charAt(j)){
                next[i] = j;
            }else{
                // 如果相同，根据next[j]跳转即可
                next[i] = next[j];
            }
        }else{
            j = next[j];
        }
    }
    return next;
}
```

i	0	1	2	3	4	5	6	7
s[i]	A	B	C	D	A	B	D	
prefix[i]	0	0	0	0	1	2	0	
next[i]	-1	0	0	0	0	1	2	0
next1[i]	-1	0	0	0	-1	0	2	-1

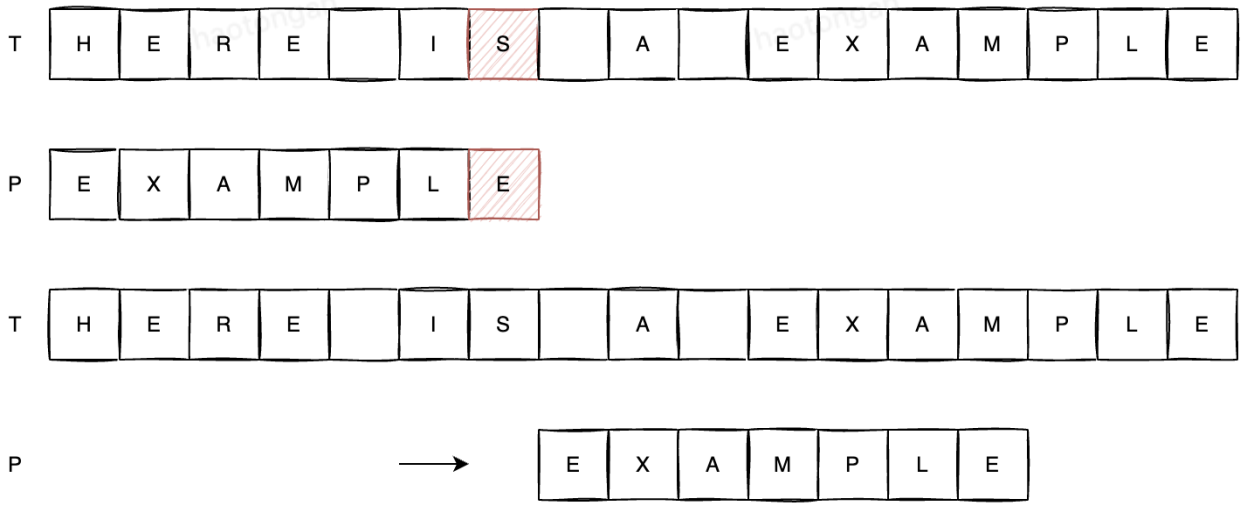
三、其他算法

这一部分，介绍几种其他字符串搜索的算法

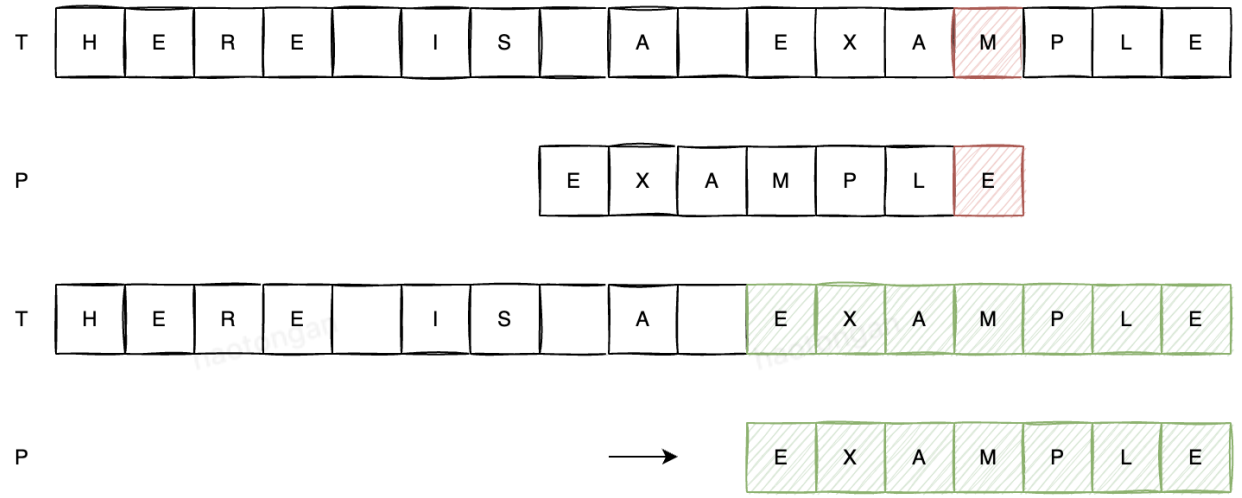
3.1 BM

1977 年，德克萨斯大学的 Robert S.Boyer 教授和 J StrotherMoore 教授发明了一种新的字符串匹配算法：Boyer-Moore算法，简称BM 算法。BM算法的基本思想是通过后缀匹配获得比前缀匹配更多的信息来实现更快的字符跳转。

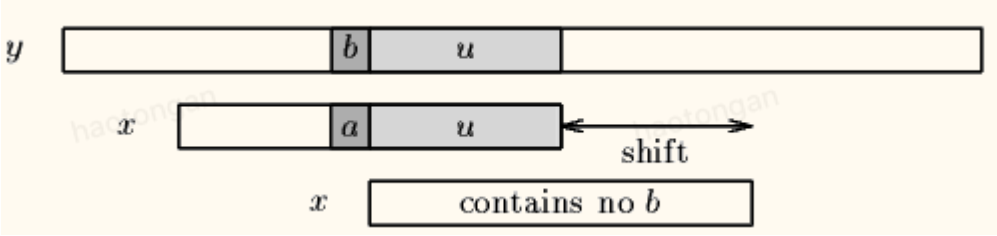
通常我们都是从左至右去匹配文本串和模式串的，下面我们从右至左尝试匹配并观察下。文本串中的字符“S”，在模式串中未出现，那么我们是不是可以跳过多余的匹配，不用去考虑模式串从文本串中第1个、第2个、第m个字符进行匹配了。可以直接将模式串向后滑动m个字符进行匹配。



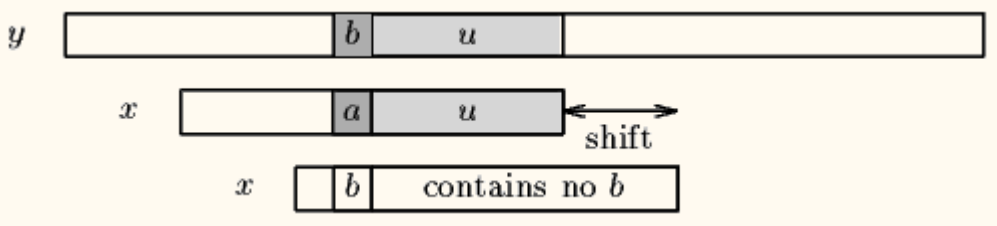
继续观察下面匹配失败的情况，我们可以发现，模式串后三个字符“E”、“L”、“P”一定无法和文本串中的字符“M”进行匹配。换句话说，直到移动到模式串中最右边的“M”(如果存在的话)之前，都是无法匹配成功的。基于这个观察，我们可以直接向后移动模式串，使最右边出现的“M”和文本串中的“M”对齐，再去继续匹配。



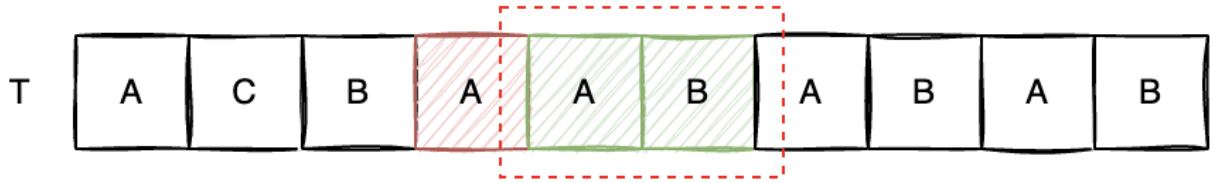
总结：当出现失配字符时（文本串的字符），如果模式串不存在该字符，则将模式串右移至失配字符的右边。

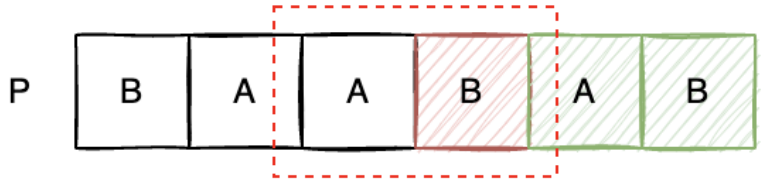


如果模式串中存在该字符，将模式串中该字符在最右边的位置，和文本串的失配字符对齐。

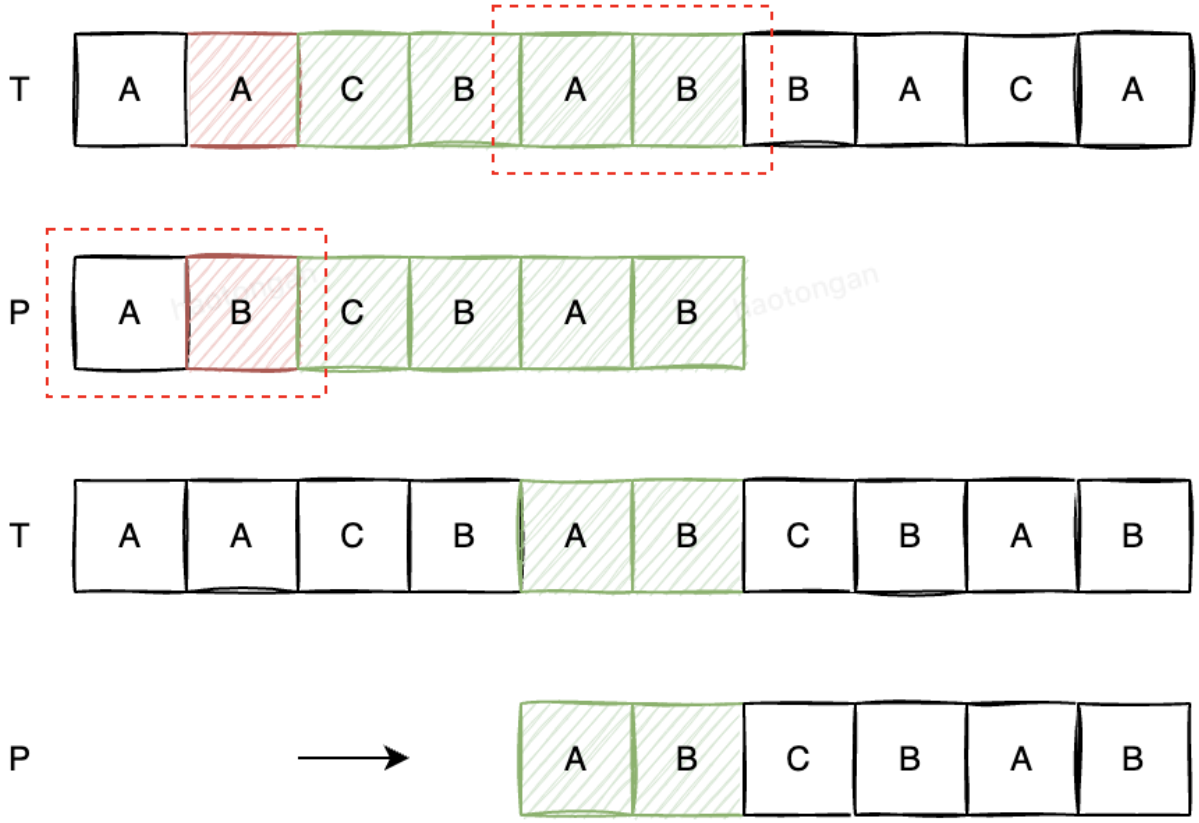


我们再观察下面的情况，我们发现文本串中字符“A”和模式串中的字符“B”匹配失败，此时已匹配的后缀“AB”我们可以在模式串中找到同样的子串“AB”，我们完全可以向后移动模式串，将两个串中的“AB”来对齐，再继续匹配。

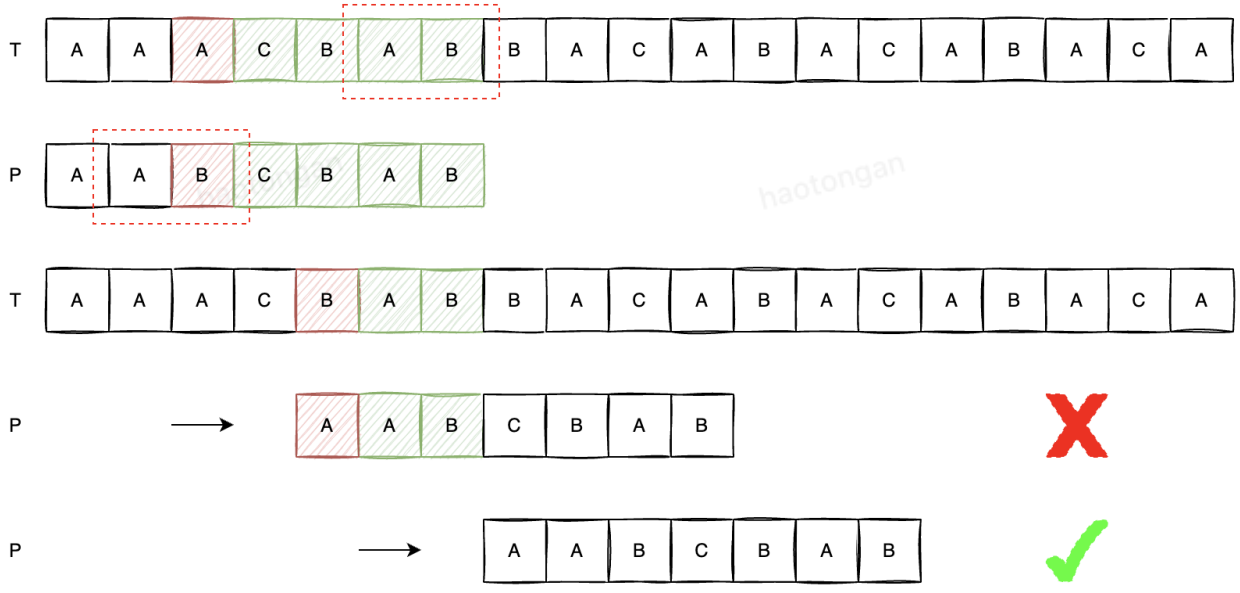




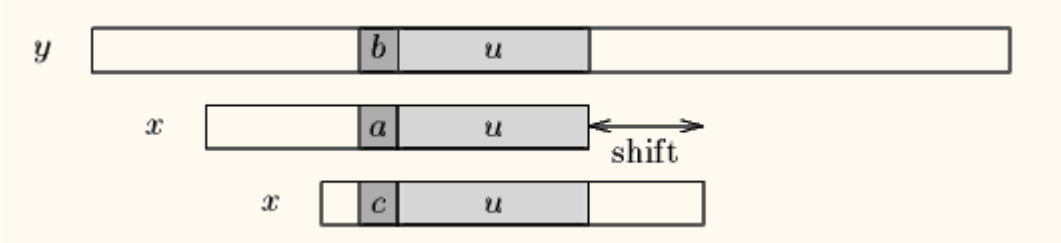
再观察下面这种情况，已经匹配的后缀“CBAB”我们无法在模式串中找到同样的部分，难道就没有办法加快匹配了吗？我们以匹配的字符串“CBAB”中的几个真后缀“BAB”、“AB”、“B”，其中“AB”作为前缀出现在了模式串中，那我们可以后移模式串，将文本串中的后缀“AB”和模式串中的前缀“AB”对齐，从而继续进行匹配。



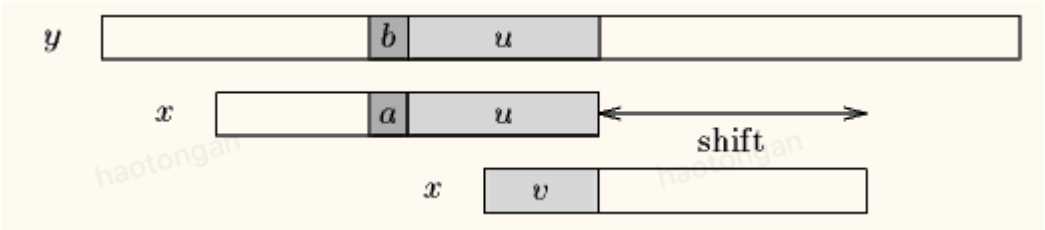
为什么已匹配的字符的真后缀必须要和模式串中的前缀匹配才可以移动呢？我们可以看下面这个例子。已匹配的“CBAB”中的真后缀“AB”，在模式串中是存在的（非前缀），那我们向后移动模式串把这两部分对齐继续匹配如何呢？这样做看似合理，但实际上却是一个无效的匹配位置。很明显，因为文本串中“AB”前的字符和模式串中“AB”前的字符一定是不匹配的，否则我们是可以找到一个比“AB”更长的匹配，且这个匹配的一定是模式串中的前缀，这就符合我们上面说的情况了。所以当没有能够匹配上合理后缀这种情况出现时，正确的移动是将模式串向后移动m位。



总结：当模式串中有子串和已匹配后缀完全相同，则将最靠右的那个子串移动到后缀的位置继续进行匹配。



如果不存在和已匹配后缀完全匹配的子串，则在已匹配后缀中找到最长的真后缀，且是模式串的前缀(t[m-s...m]=P[0...s])



如果完全不存在和好后缀匹配的子串，则右移整个模式串。

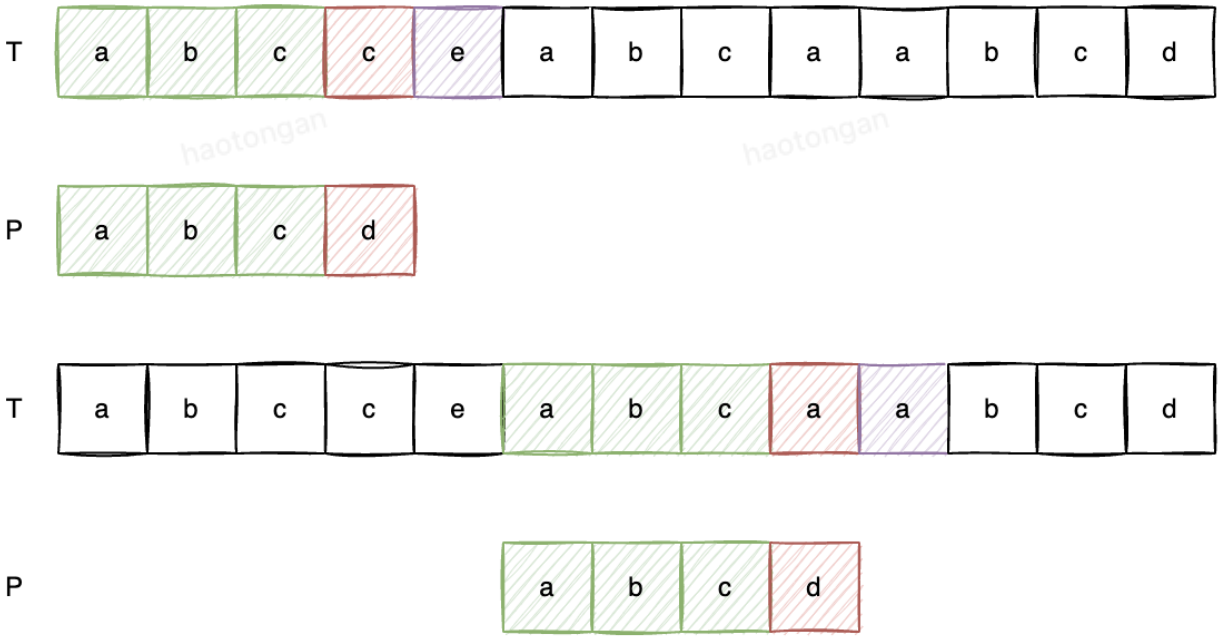
BM算法在实际匹配时，考虑上面两种策略，当匹配失败发生时，会选择能够移动的最大的距离，来去移动模式串，从而加速匹配。实际情况，失配字符移动策略已经能很好的加速匹配过程，因为模式串本身字符数量是要少于文本串的，Quick Search algorithm（Sunday）正是利用这一策略的算法（有些许不同），或者说是一种简化版的BM算法。

3.2 Sunday

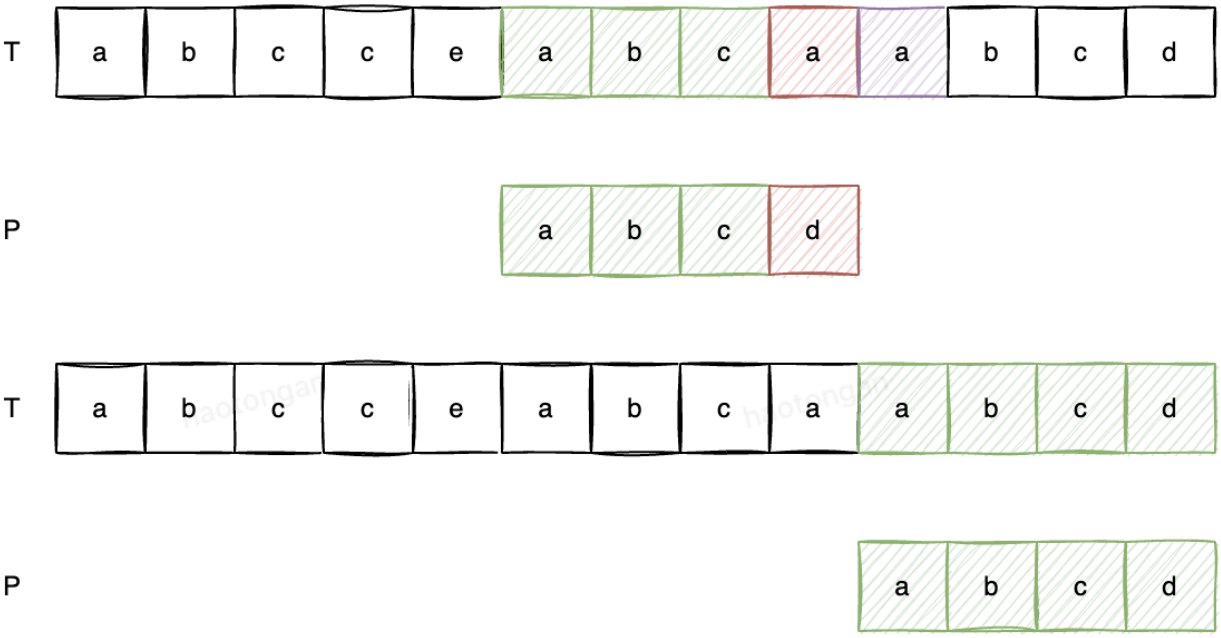
Sunday 算法是 Daniel M.Sunday 于 1990 年提出的字符串模式匹配。其效率在匹配随机的字符串时比其他匹配算法还要更快。Sunday 算法的实现可比 KMP，BM 的实现容易的多。

Sunday算法思想跟BM算法很相似，在匹配失败时关注的是文本串中参加匹配的最末位字符的下一位字符。如果该字符没有在模式串中出现则直接跳过，即移动步长= 模式串长度+1；否则，同BM算法一样其移动步长=模式串中最右端的该字符到末尾的距离+1。

文本串T中字符“c”和模式串中的字符“d”不匹配。我们观察文本串参与匹配的末位的下一个字符“e”，可以知道“e”没有出现在模式串中。于是移动模式串长度+1。



继续匹配，我们发现文本串T中字符“a”和模式串中的字符“d”不匹配。我们观察文本串参与匹配的末位的下一个字符“a”，可以知道“a”出现在模式串中（最右的位置）。于是移动模式串该字符到末尾的距离+1。



3.3 Rabin-Karp

Rabin-Karp 算法，由 Richard M. Karp 和 Michael O. Rabin 在 1987 年发表，它也是用来解决多模式串匹配问题的。该算法实现方式与上述的字符匹配不同，首先是计算两个字符串的哈希值，然后通过比较这两个哈希值的大小来判断是否出现匹配。

为了帮助更好的解决字符串匹配问题，哈希函数应该具有以下属性:

1.高效的、可计算的

2.更好的识别字符串

3.在计算hash(y[j+1 ..j+m])应该可以容易的从hash(y[j..j+m-1])和y[j+m]中得到结果，即hash(y[j+1 ..j+m])=rehash(y[j],y[j+m],hash(y[j..j+m-1])

我们定义hash函数如下：

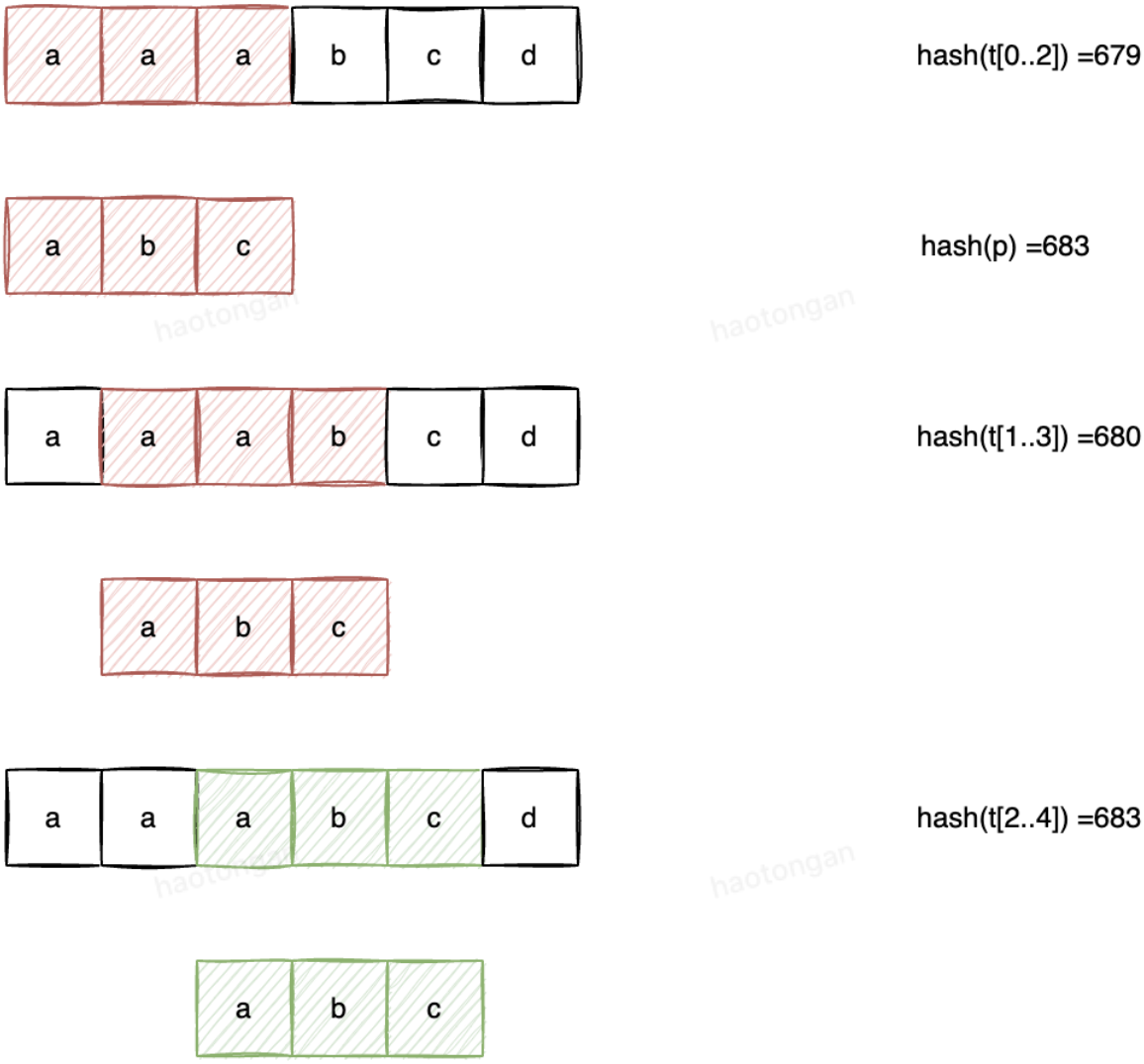
hash(w[0 ..m-1])=(w[0]*2^m-1+w[1]*2^m-2+…+w[m-1]*2^0) mod q

由于计算的hash值可能会很大，所以需要取模操作，q最好选取一个比较大的数，且是一个质数，w[i]表示y[i]对应的ASCII码。

hash(w[1..m])=rehash(w[0],w[m],hash(w[0..m-1]))

rehash(a,b,h)= ((h-a*2^m-1)*2+b) mod q

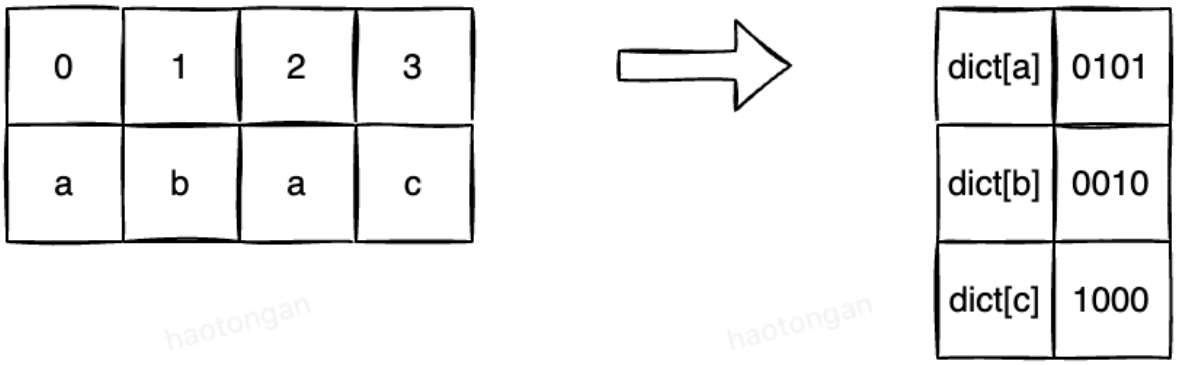
匹配过程中，不断滑动窗口来计算文本串的hash值和模式串的是否相同，当出现相同时，还需要再检查一遍字符串是否真正相同，因为会出现哈希碰撞的情况。



3.4 Shift-and/or

Shift-and算法的总体思路是把模式串预处理成一种特殊编码形式，然后根据这种编码形式去逐位匹配文本串。

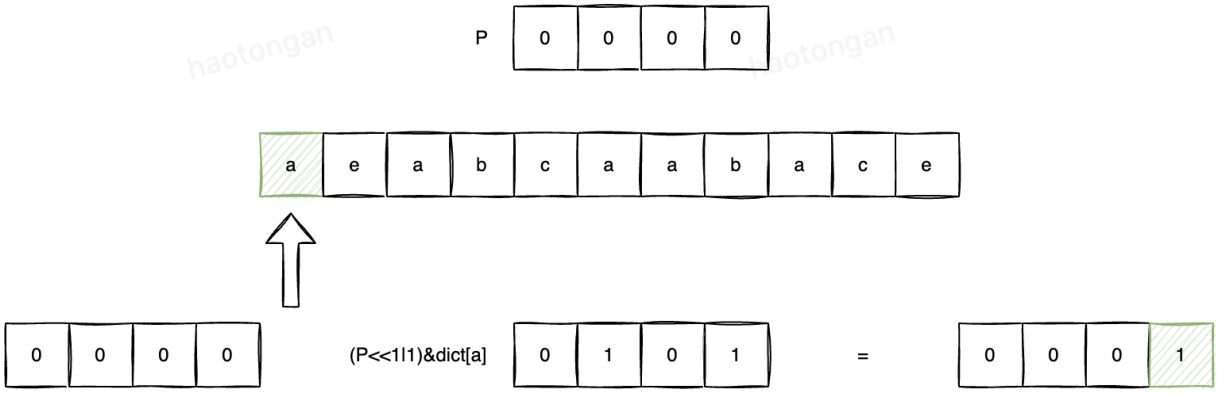
首先对模式串进行预处理，利用二进制数位进行编码。如果模式串为“abac”，a出现在第0位和第2位，那么则可以保存a的信息为5(二进制为0101)，同样的，我们把模式串出现的所有字符均用这种方式编码，并保存起来。



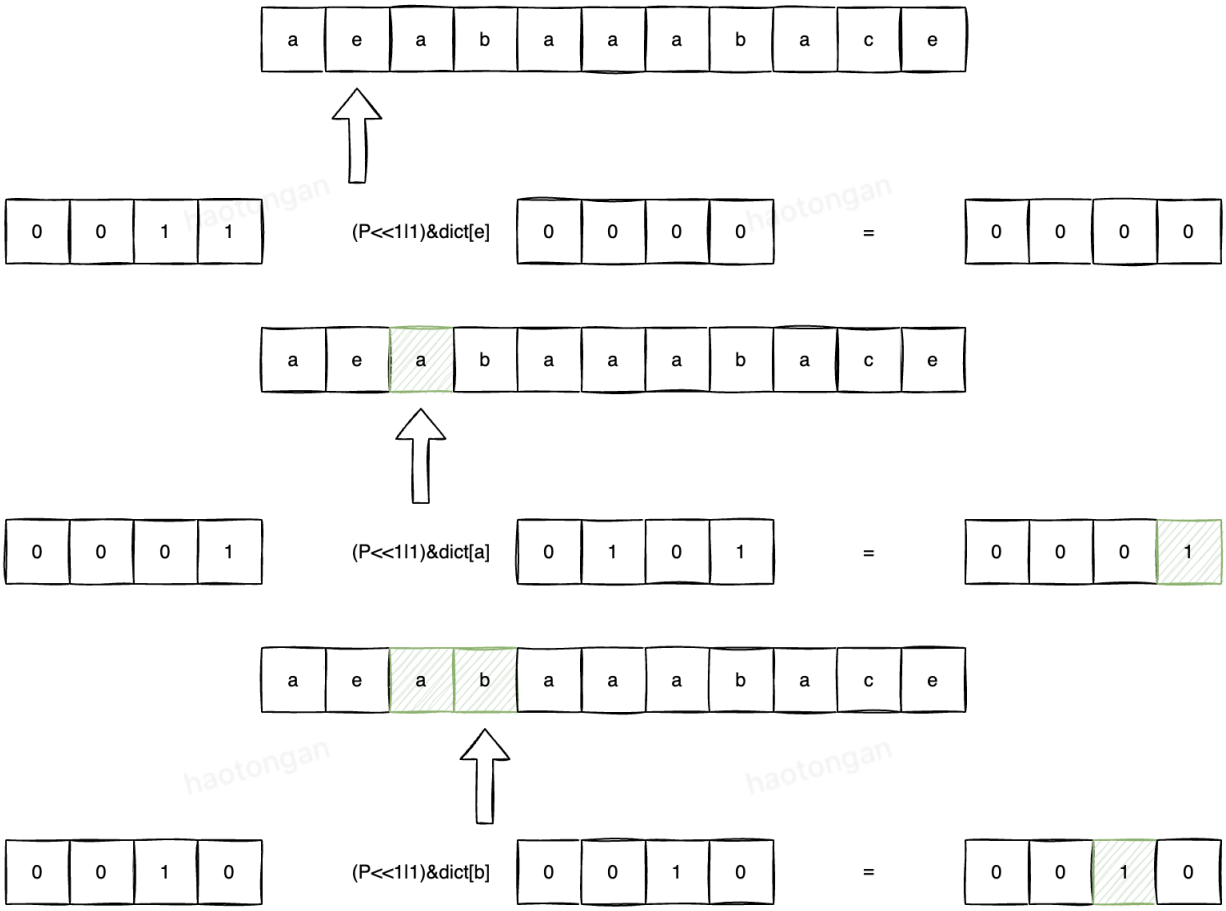
对于每一位文本串字符，我们定义一个对应的状态码数字P，当P[i]=1时，则表示以这一位文本串为末尾时，能和模式串的第0位到第i位的字符能完全匹配。我们看一下具体的匹配过程。

文本串“aeabcaabace”和模式串“abac”，初始化P=0，遍历文本串中的每一个字符，同时根据存储的字符编码信息，来更新匹配结果，也就是状态码P。

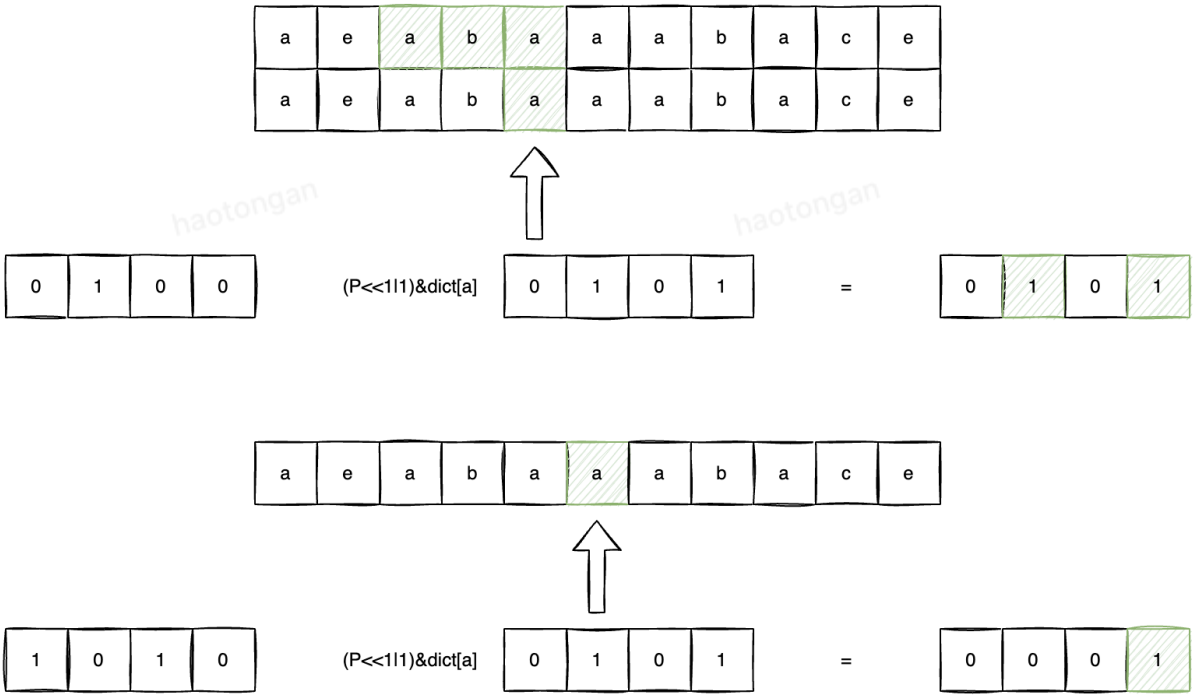
在第一次计算完成后，状态码P=0001，根据我们上面的定义，P[0]=1即表示以这一位文本串为末尾，模式串中的第0位到第0位的字符是匹配的。



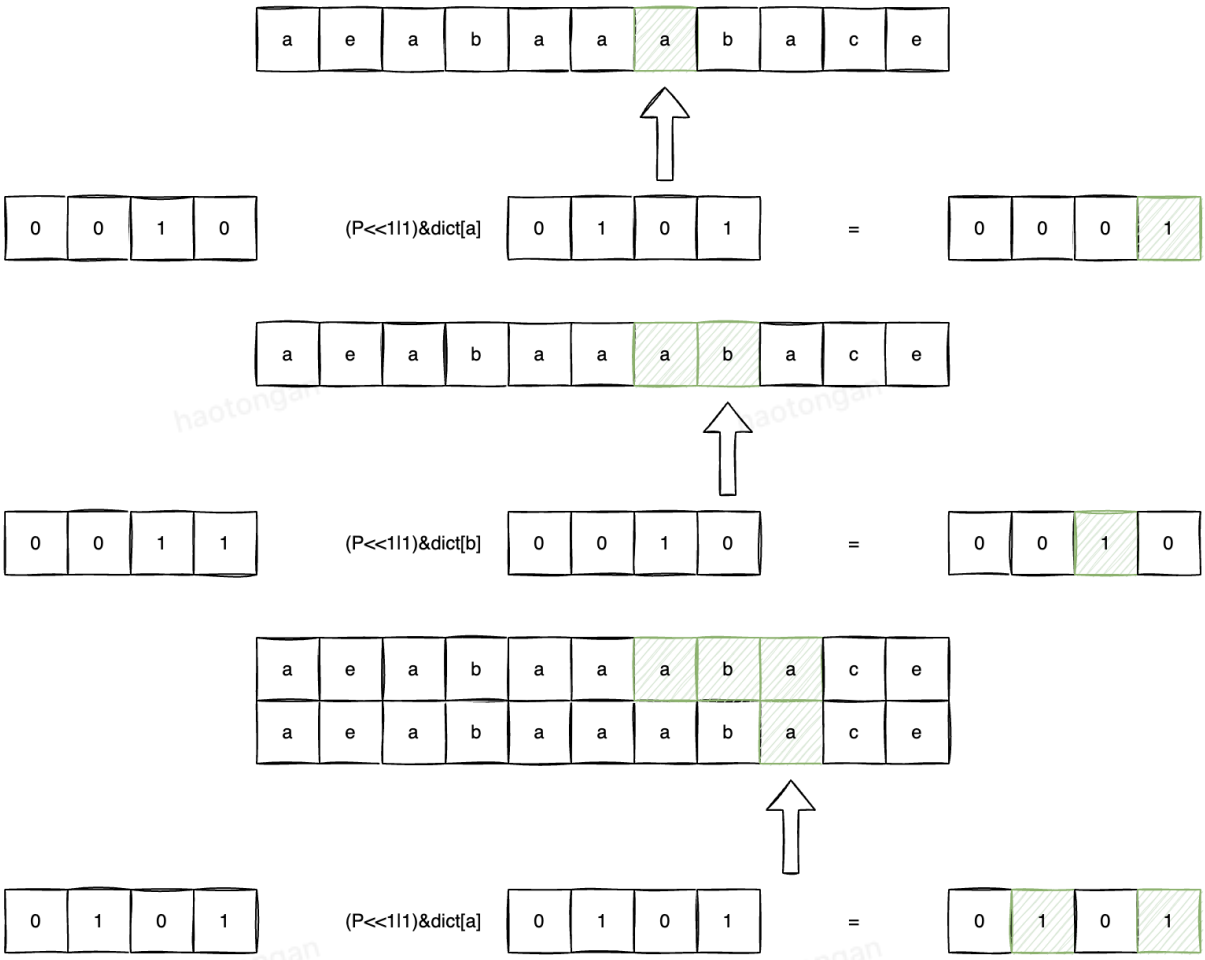
进行完一次匹配后，P左移一位，将第0位置1，同时和对应字符的编码进行&操作（即尝试匹配该字符），更新状态码P。



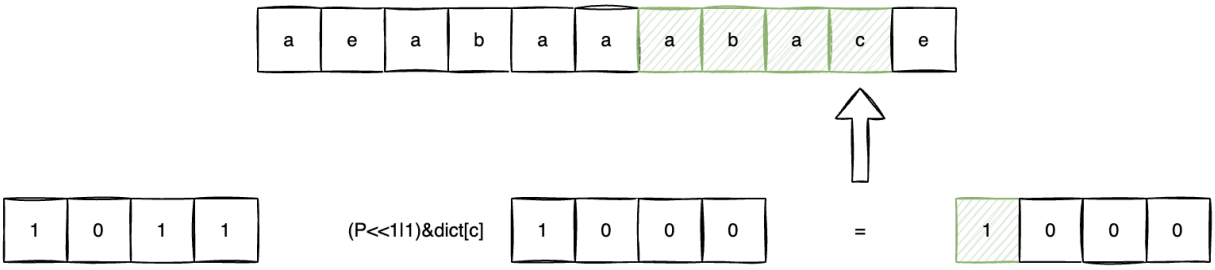
可以看到当状态码P=0101时，P[2]=1表示当前字符匹配了模式串p[0..2]=“aba”，P[0]=1表示当前字符匹配了模式串p[0..0]=“a”，也就是说，状态码P是能够存储多种部分匹配的结果。



继续匹配



当P=1000时，也就是说P[3]=1即匹配模式串p[0...3]=“abac”，正好找到了一个对应的匹配，而我们也可以根据此条件来判断是否已经找到了匹配。



Shift-and使用的二进制信息来编码模式串，使用位运算&来达到并行匹配字符串，利用状态码P来保存当前位的匹配结果。可以观察出算法的时间复杂度很低，如果模式串的长度不超过机器字长，其效率是非常高的。

Shift-or在这里就不多做介绍了，其原理和Shift-and类似，只不过Shift-or使用0来标识存在，同时使用｜来代替&进行状态码的计算。

相关参考：

1.http://igm.univ-mlv.fr/~lecroq/string/node8.html#SECTION0080

2.http://igm.univ-mlv.fr/~lecroq/string/node14.html#SECTION00140

3.https://shanire.gitee.io/oiwiki/string/kmp/#knuth-morris-pratt

4.https://shanire.gitee.io/oiwiki/string/bm/

5.http://igm.univ-mlv.fr/~lecroq/string/node6.html#SECTION0060
6.https://baike.baidu.com/item/sunday%20%E7%AE%97%E6%B3%95/1816405
7.http://igm.univ-mlv.fr/~lecroq/string/node5.html#SECTION0050