如果你看不懂 KMP 算法，那就看一看这篇文章(绝对原创，绝对通俗易懂)

时间 2014-03-09 20:32:21 CSDN 博客
原文 http://blog.csdn.net/u011564456/article/details/20862555
主题 算法

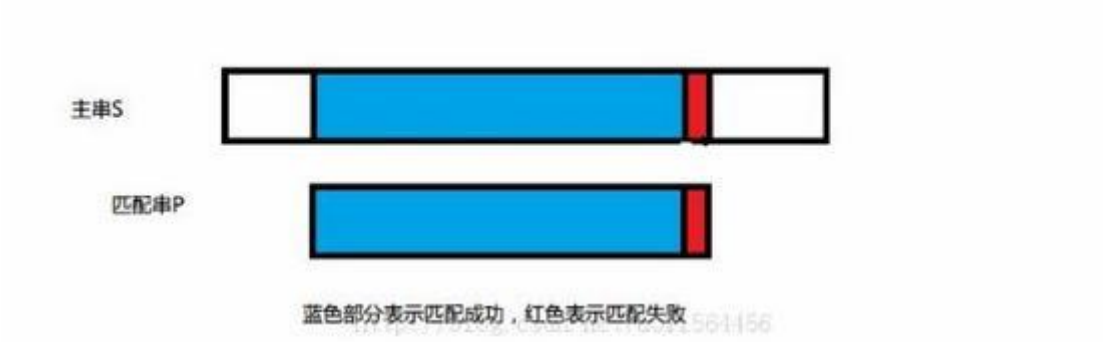
如果你看不懂 KMP 算法，那就看一看这篇文章 (绝对原创，绝对通俗易懂)

KMP 算法，俗称“看毛片”算法，是字符串匹配中的很强大的一个算法，不过，对于初学者来说，要弄懂它确实不易。整个寒假，因为家里没有网，为了理解这个算法，那可是花了九牛二虎之力！不过，现在我基本上对这个算法理解算是比较透彻了！特写此文与大家分享分享！

我个人总结了， KMP 算法之所以难懂，很大一部分原因是很多实现的方法在一些细节的差异。怎么说呢，举我寒假学习的例子吧，我是看了一种方法后，似懂非懂，然后去看另外的方法，就全都乱了！体现在几个方面： next 数组，有的叫做“失配函数”，其实是一个东西； next 数组中，有的是以下标为 0 开始的，有的是以 1 开始的； KMP 主算法中，当发生失配时，取的 next 数组的值也不一样！就这样，各说各的，乱的很！

所以，在阐述我的理解之前，我有必要说明一下，我是用 next 数组的， next 数组是以下标 0 开始的！还有，我不会在一些基础的概念上浪费太多，所以你在看这篇文章时必须要懂得一些基本的概念，例如 “ 朴素字符串匹配 ”“ 前缀 ”， “ 后缀 ” 等！还有就是，这篇文章的每一个字都是我辛辛苦苦码出来的，图也是我自己画的！如果要转载，请注明出处！好了，开始吧！

假设在我们的匹配过程中出现了这一种情况：

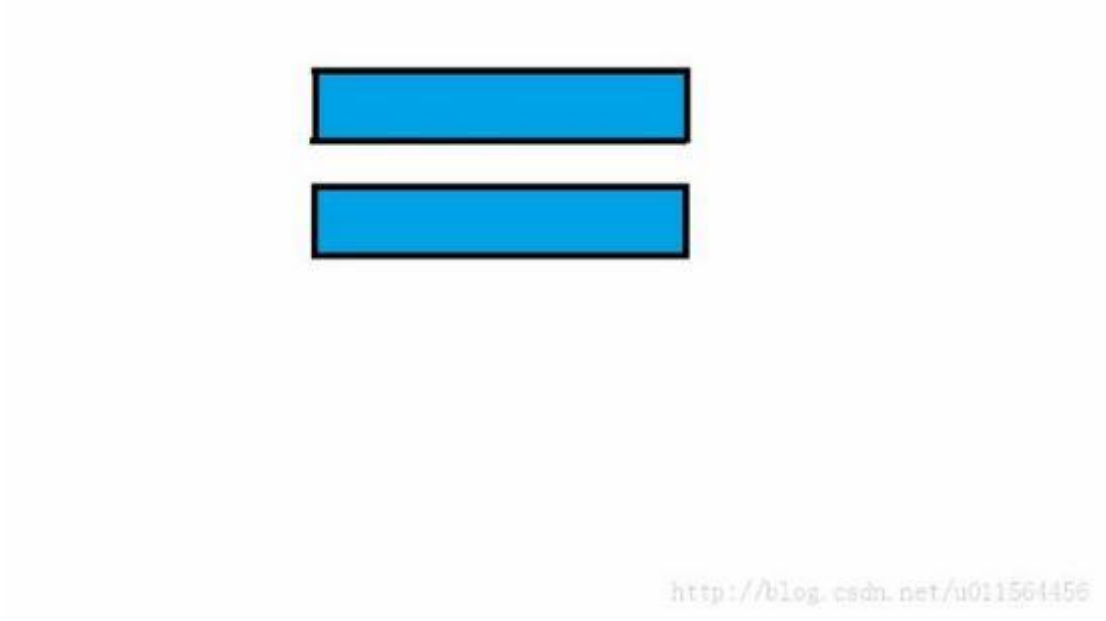


根据 KMP 算法，在该失配位会调用该位的 next 数组的值！在这里有必要来说一下 next 数组的作用！说的太繁琐怕你听不懂，让我用一句话来说明：

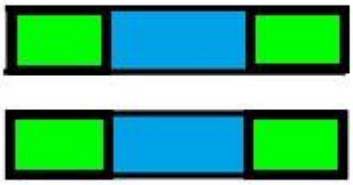
返回失配位之前的最长公共前后缀！

好，不管你懂不懂这句话，我下面的文字和图应该会让你懂这句话的意思以及作用的！

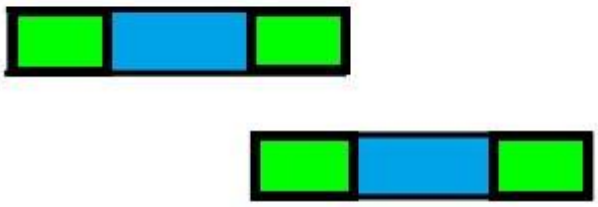
首先，我们取之前已经匹配的部分（即蓝色的那部分！）



我们在上面说到 next 数组的作用时，说到“ 最长公共前后缀 ”，体现到图中就是这个样子！

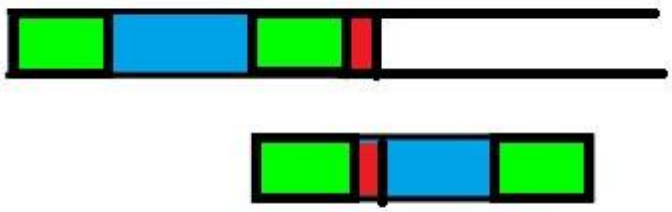


接下来，就是最重要的了！



没错，这个就是 next 数组的作用了：

返回当前的最长公共前后缀长度，假设为 len 。因为数组是由 0 开始的，所以 next 数组让第 len 位与主串匹配就是拿最长前缀之后的第 1 位与失配位重新匹配，避免匹配串从头开始！如下图所示！



（重新匹配刚才的失配位！）

如果都说成这样你都不明白，那么你真的得重新理解什么是 KMP 算法了！

接下来最重要的，也是 KMP 算法的核心所在，就是 next 数组的求解！不过，在这里我找到了一个全新的理解方法！如果你懂的上面我写的的，那么下面的内容你只需稍微思考一下就行了！

跟刚才一样，我用一句话来阐述一下 next 数组的求解方法，其实也就是两个字：

继承

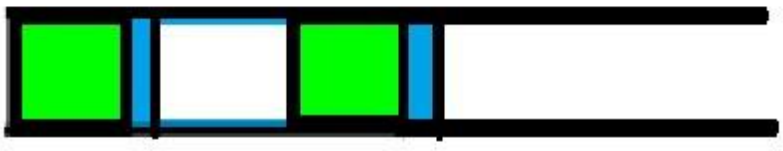
a 、当前面字符的前一个字符的对称程度为 0 的时候，只要将当前字符与子串第一个字符进行比较。这个很好理解啊，前面都是 0 ，说明都不对称了，如果多加了一个字符，要对称的话最多是当前的和第一个对称。比如 agcta 这个里面 t 的是 0 ，那么后面的 a 的对称程度只需要看它是不是等于第一个字符 a 了。

b 、按照这个推理，我们就可以总结一个规律，不仅前面是 0 呀，如果前面一个字符的 next 值是 1 ，那么我们就把当前字符与子串第二个字符进行比较，因为前面的是 1 ，说明前面的字符已经和第一个相等了，如果这个又与第二个

相等了，说明对称程度就是 2 了。有两个字符对称了。比如上面 agctag ，倒数第二个 a 的 next 是 1 ，说明它和第一个 a 对称了，接着我们就把最后一个 g 与第二个 g 比较，又相等，自然对称成都就累加了，就是 2 了。

c 、按照上面的推理，如果一直相等，就一直累加，可以一直推啊，推到这里应该一点难度都没有吧，如果你觉得有难度说明我写的太失败了。

当然不可能会那么顺利让我们一直对称下去，如果遇到下一个不相等了，那么说明不能继承前面的对称性了，这种情况只能说明没有那么多对称了，但是不能说明一点对称性都没有，所以遇到这种情况就要重新来考虑，这个也是难点所在。



如果蓝色的部分相同，则当前 next 数组的值为上一个 next 的值加一，如果不相同，就是我们下面要说的！

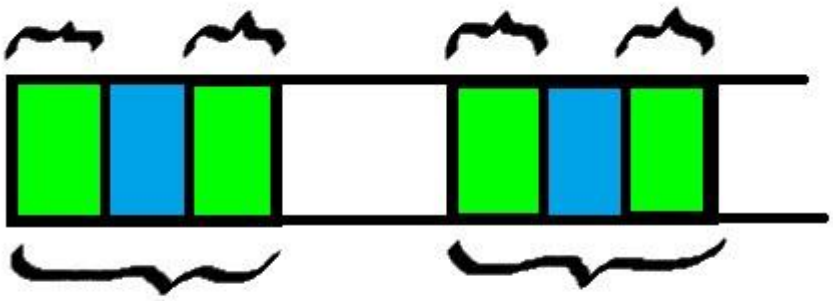
如果不相同，用一句话来说，就是：

从前面来找子前后缀

1 、如果要存在对称性，那么对称程度肯定比前面这个的对称程度小，所以要找个更小的对称，这个不用解释了吧，如果大那么就继承前面的对称性了。

2 、要找更小的对称，必然在对称内部还存在子对称，而且这个必须紧接着在子对称之后。

如果看不懂，那么看一下图吧！



好了，我已经把该说的尽可能以最浅显的话和最直接的图展示出来了，如果还是不懂，那我真的没有办法了！

说了这么多，下面是代码实现

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define N 100



void cal_next( char * str, int * next, int len )

{

  int i, j;



  next[0] = -1;

  for( i = 1; i < len; i++ )
```

```
  {
    j = next[ i - 1 ];
    while( str[ j + 1 ] != str[ i ] && ( j >= 0 ) )
    {
      j = next[ j ];
    }
    if( str[ i ] == str[ j + 1 ] )
    {
      next[ i ] = j + 1;
    }
    else
    {
      next[ i ] = -1;
    }
  }
}


int KMP( char * str, int slen, char * ptr, int plen, int * next )
{
```

```
int s_i = 0, p_i = 0;



while( s_i < slen && p_i < plen )

{

  if( str[ s_i ] == ptr[ p_i ] )

  {

    s_i++;

    p_i++;

  }

  else

  {

    if( p_i == 0 )

    {

      s_i++;

    }

    else

    {

      p_i = next[ p_i - 1 ] + 1;

    }
```

```
    }
  }
  return ( p_i == plen ) ? ( s_i - plen ) : -1;
}


int main()
{
  char str[ N ] = {0};
  char ptr[ N ] = {0};
  int slen, plen;
  int next[ N ];

  while( scanf( "%s%s", str, ptr ) )
  {
    slen = strlen( str );
    plen = strlen( ptr );
    cal_next( ptr, next, plen );
    printf( "%d\n", KMP( str, slen, ptr, plen, next ) );
  }
```

```
  return 0;


}
```

如果有什么问题，欢迎评论指正！还是大一新手，很需要进步!

【经典算法】——KMP，深入讲解 next 数组的求解

前言

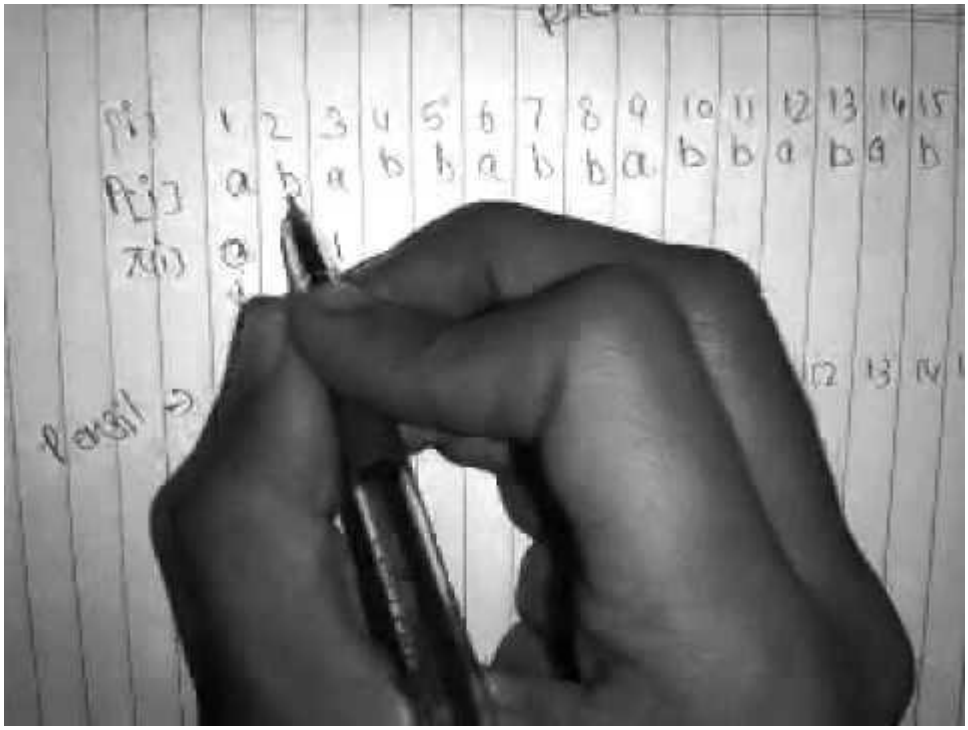
之前对 kmp 算法虽然了解它的原理，即求出 P_0・・・P_i 的最大相同前后缀长度 k；但是问题在于如何求出这个最大前后缀长度呢？我觉得网上很多帖子都说的不是很清楚，总感觉没有把那层纸戳破，后来翻看算法导论，32 章 字符串匹配虽然讲到了对前后缀计算的正确性，但是大量的推理证明不大好理解，没有与程序结合起来讲。今天我在这里讲一讲我的一些理解，希望大家多多指教，如果有不清楚的或错误的请给我留言。

1. kmp 算法的原理：

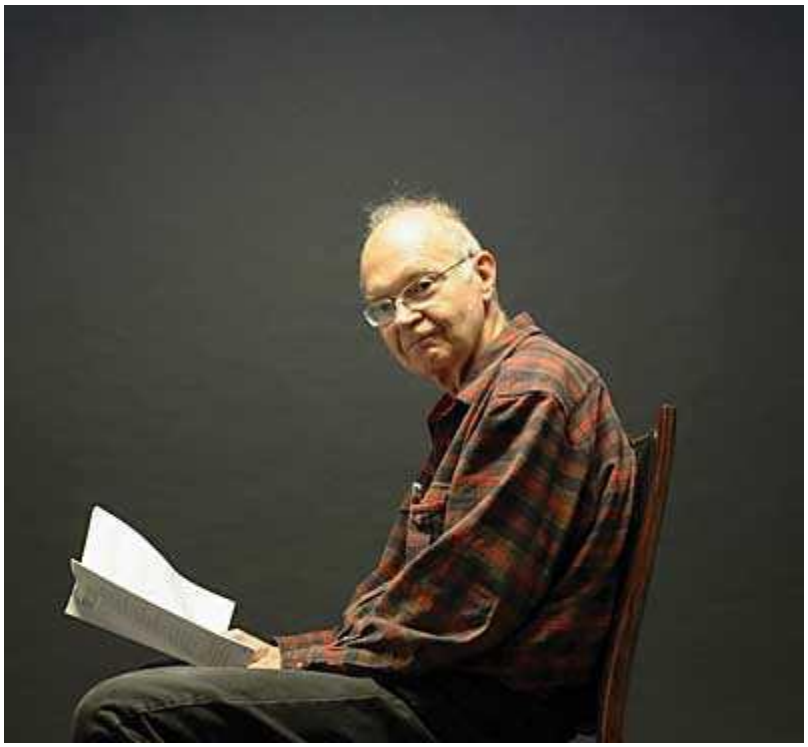
本部分内容转自：
http://www.ruanyifeng.com/blog/2013/05/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm.html

字符串匹配是计算机的基本任务之一。

举例来说，有一个字符串"BBC ABCDAB ABCDABCDABDE"，我想知道，里面是否包含另一个字符串"ABCDABD"？

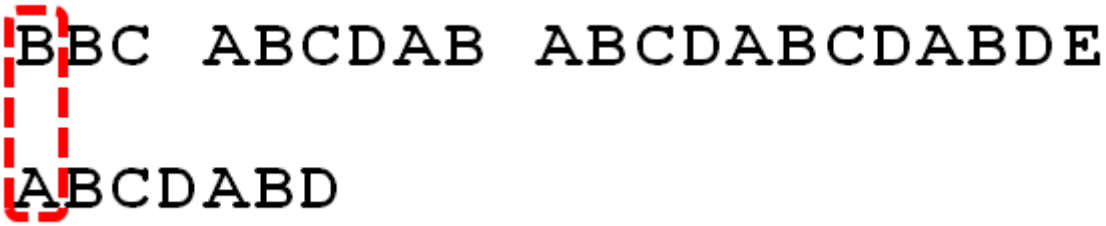


许多算法可以完成这个任务，Knuth-Morris-Pratt 算法（简称 KMP）是最常用的之一。它以三个发明者命名，起头的那个 K 就是著名科学家 Donald Knuth。



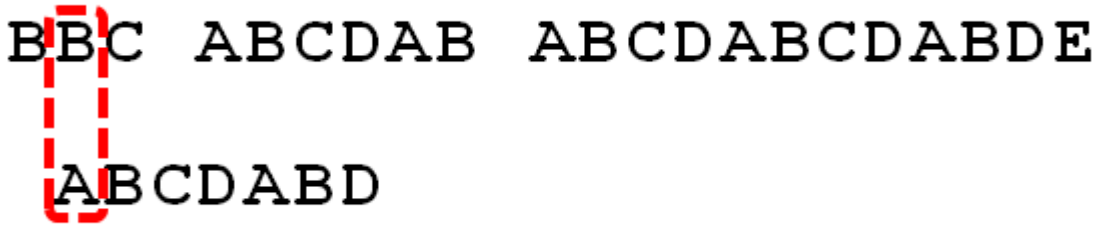
这种算法不太容易理解，网上有很多解释，但读起来都很费劲。直到读到 Jake Boxer 的文章，我才真正理解这种算法。下面，我用自己的语言，试图写一篇比较好懂的 KMP 算法解释。

1.



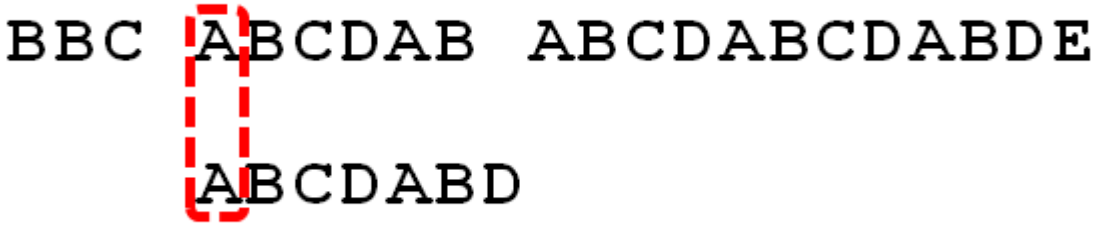
首先，字符串"BBC ABCDAB ABCDABCDABDE"的第一个字符与搜索词"ABCDABD"的第一个字符，进行比较。因为 B 与 A 不匹配，所以搜索词后移一位。

2.



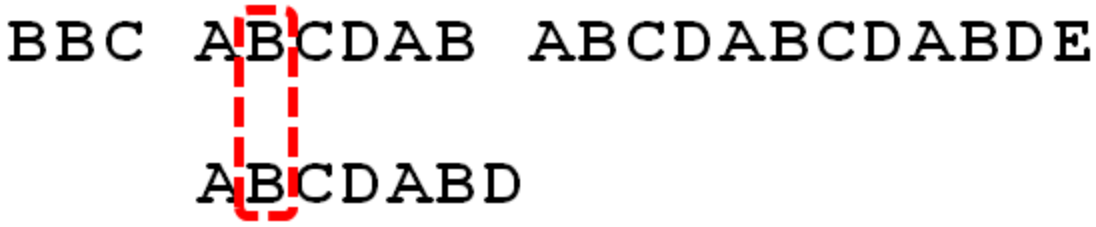
因为 B 与 A 不匹配，搜索词再往后移。

3.



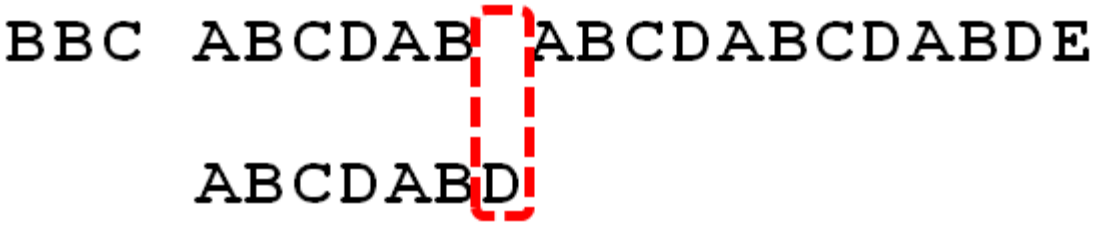
就这样，直到字符串有一个字符，与搜索词的第一个字符相同为止。

4.



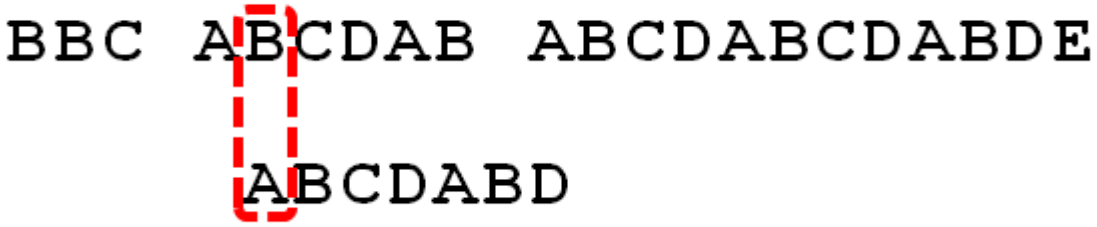
接着比较字符串和搜索词的下一个字符，还是相同。

5.



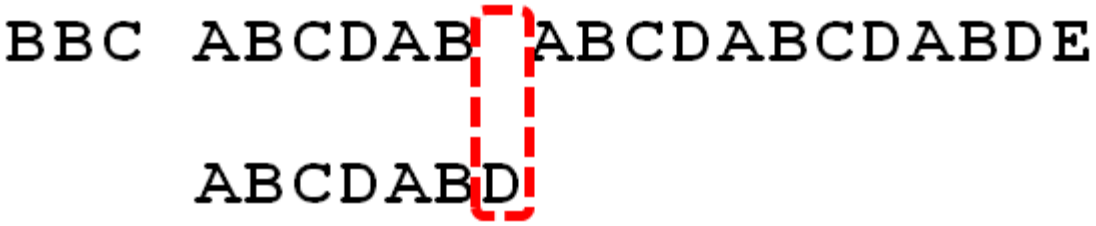
直到字符串有一个字符，与搜索词对应的字符不相同为止。

6.



这时，最自然的反应是，将搜索词整个后移一位，再从头逐个比较。这样做虽然可行，但是效率很差，因为你要把"搜索位置"移到已经比较过的位置，重比一遍。

7.



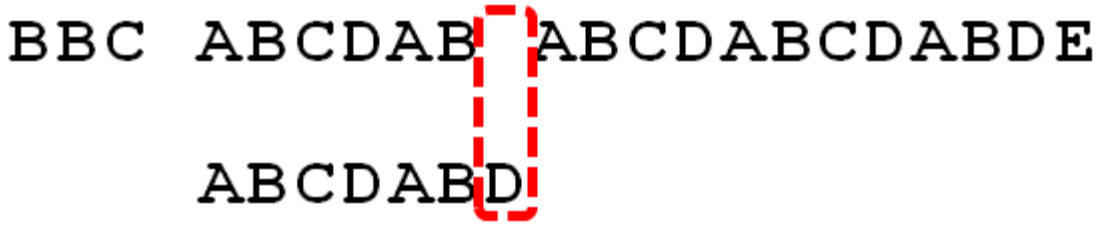
一个基本事实是，当空格与 D 不匹配时，你其实知道前面六个字符是"ABCDAB"。KMP 算法的想法是，设法利用这个已知信息，不要把"搜索位置"移回已经比较过的位置，继续把它向后移，这样就提高了效率。

8.

搜索词	A	B	C	D	A	B	D
部分匹配值	0	0	0	0	1	2	0

怎么做到这一点呢？可以针对搜索词，算出一张《部分匹配表》（Partial Match Table）。这张表是如何产生的，后面再介绍，这里只要会用就可以了。

9.

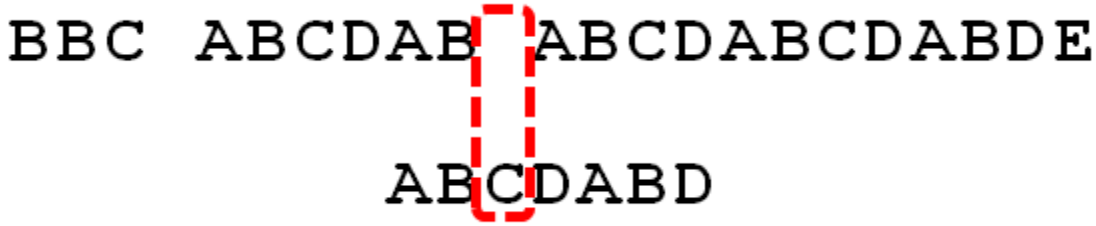


已知空格与 D 不匹配时，前面六个字符"ABCDAB"是匹配的。查表可知，最后一个匹配字符 B 对应的"部分匹配值"为 2，因此按照下面的公式算出向后移动的位数：

移动位数 = 已匹配的字符数 - 对应的部分匹配值

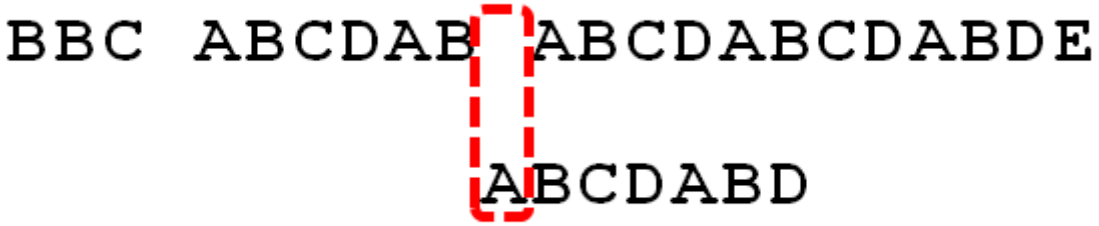
因为 6 - 2 等于 4，所以将搜索词向后移动 4 位。

10.



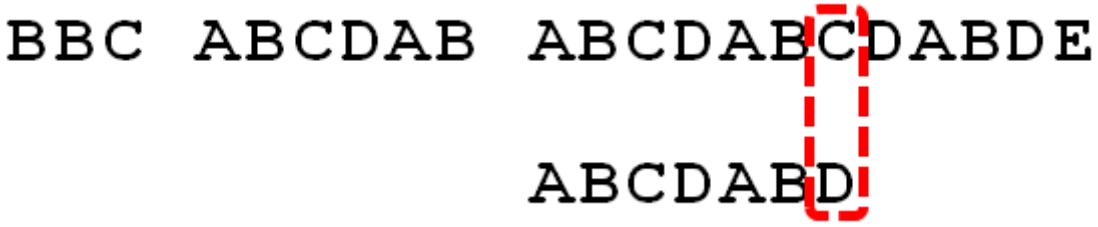
因为空格与 C 不匹配，搜索词还要继续往后移。这时，已匹配的字符数为 2（"AB"），对应的"部分匹配值"为 0。所以，移动位数 = 2 - 0，结果为 2，于是将搜索词向后移 2 位。

11.



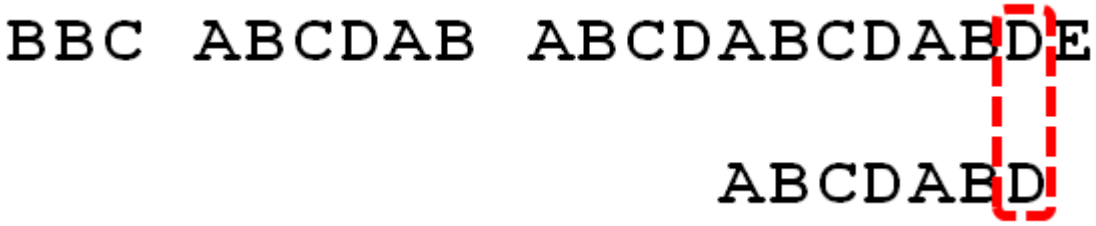
因为空格与 A 不匹配，继续后移一位。

12.



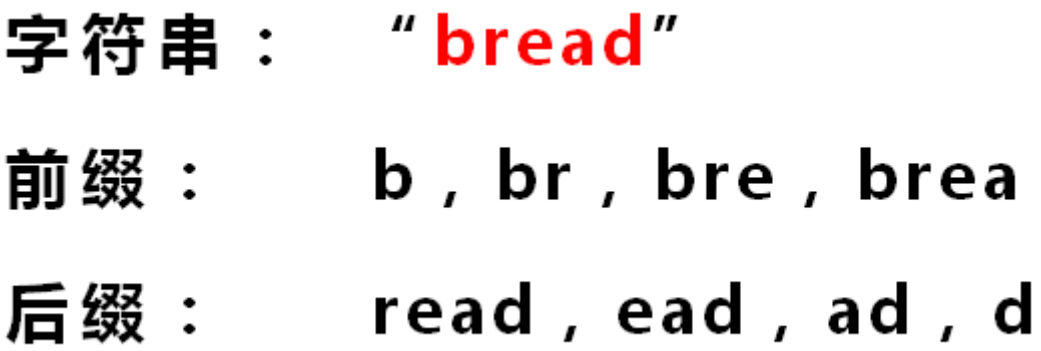
逐位比较，直到发现 C 与 D 不匹配。于是，移动位数 = 6 - 2，继续将搜索词向后移动 4 位。

13.



逐位比较，直到搜索词的最后一位，发现完全匹配，于是搜索完成。如果还要继续搜索（即找出全部匹配），移动位数 = 7 - 0，再将搜索词向后移动 7 位，这里就不再重复了。

14.



下面介绍《部分匹配表》是如何产生的。

首先，要了解两个概念："前缀"和"后缀"。 "前缀"指除了最后一个字符以外，一个字符串的全部头部组合；"后缀"指除了第一个字符以外，一个字符串的全部尾部组合。

15.

搜索词	A	B	C	D	A	B	D
部分匹配值	0	0	0	0	1	2	0

"部分匹配值"就是"前缀"和"后缀"的最长的共有元素的长度。以"ABCDABD"为例，

- "A"的前缀和后缀都为空集，共有元素的长度为 0；

- "AB"的前缀为[A]，后缀为[B]，共有元素的长度为 0；

- "ABC"的前缀为[A, AB]，后缀为[BC, C]，共有元素的长度 0；

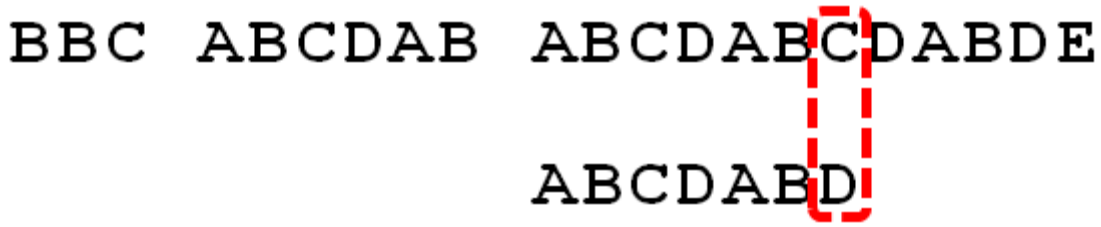
- "ABCD"的前缀为[A, AB, ABC]，后缀为[BCD, CD, D]，共有元素的长度为 0；

- "ABCDA"的前缀为[A, AB, ABC, ABCD]，后缀为[BCDA, CDA, DA, A]，共有元素为"A"，长度为 1；

- "ABCDAB"的前缀为[A, AB, ABC, ABCD, ABCDA]，后缀为[BCDAB, CDAB, DAB, AB, B]，共有元素为"AB"，长度为 2；

- "ABCDABD"的前缀为[A, AB, ABC, ABCD, ABCDA, ABCDAB]，后缀为[BCDABD, CDABD, DABD, ABD, BD, D]，共有元素的长度为 0。

16.



"部分匹配"的实质是，有时候，字符串头部和尾部会有重复。比如，"ABCDAB"之中有两个"AB"，那么它的"部分匹配值"就是 2（"AB"的长度）。搜索词移动的时候，第一个"AB"向后移动 4 位（字符串长度-部分匹配值），就可以来到第二个"AB"的位置。

2.next 数组的求解思路

通过上文完全可以对 kmp 算法的原理有个清晰的了解，那么下一步就是编程实现了，其中最重要的就是如何根据待匹配的模版字符串求出对应每一位的最大相同前后缀的长度。我先给出我的代码：

```
1 void makeNext(const char P[],int next[])
2 {
3     int q,k;//q:模版字符串下标；k:最大前后缀长度
4     int m = strlen(P);//模版字符串长度
5     next[0] = 0;//模版字符串的第一个字符的最大前后缀长度为 0
6     for (q = 1,k = 0; q < m; ++q)//for 循环，从第二个字符开始，依次计算每一个字符对应的 next 值
7     {
8         while(k > 0 && P[q] != P[k])//递归的求出 P[0]・・・P[q]的最大的相同的前后缀长度 k
```

```
 9              k = next[k-1];          //不理解没关系看下面的分析，这个 while 循环是整段代码的精髓所在，确实不好理解
10          if (P[q] == P[k])//如果相等，那么最大相同前后缀长度加 1
11          {
12              k++;
13          }
14          next[q] = k;
15      }
16 }
```

现在我着重讲解一下 while 循环所做的工作：

1. 已知前一步计算时最大相同的前后缀长度为 k（k>0），即 P[0]…P[k-1]；
2. 此时比较第 k 项 P[k]与 P[q],如图 1 所示
3. 如果 P[K]等于 P[q]，那么很简单跳出 while 循环;
4. **关键！关键有木有！关键如果不等呢？？？**那么我们应该利用已经得到的 next[0]…next[k-1]来**求 P[0]…P[k-1]这个子串中最大相同前后缀**，可能有同学要问了——为什么要求 P[0]…P[k-1]的最大相同前后缀呢？？？是啊！为什么呢？ **原因**在于 P[k]已经和 P[q]失配了，而且 P[q-k] … P[q-1]又与 P[0] …P[k-1]相同，看来 P[0]…P[k-1]这么长的子串是用不了了，那么我要找个同样也是 P[0]打头、P[k-1]结尾的子串即 P[0]…P[j-1](j==next[k-1])，看看它的下一项 P[j]是否能和 P[q]匹配。如图 2 所示

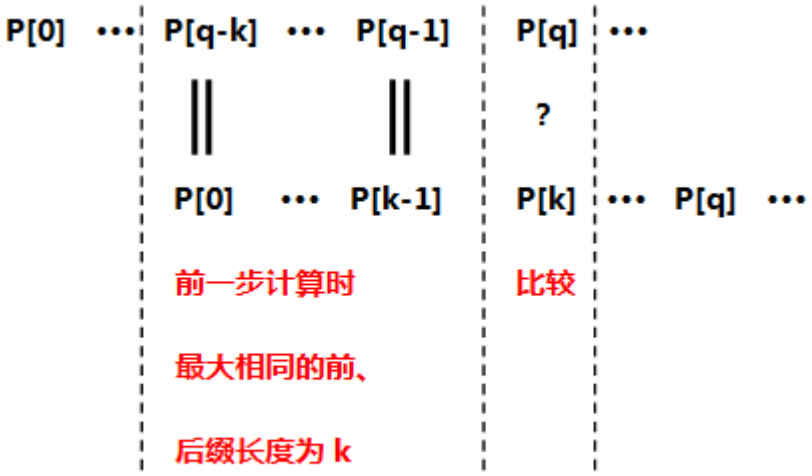


图 1

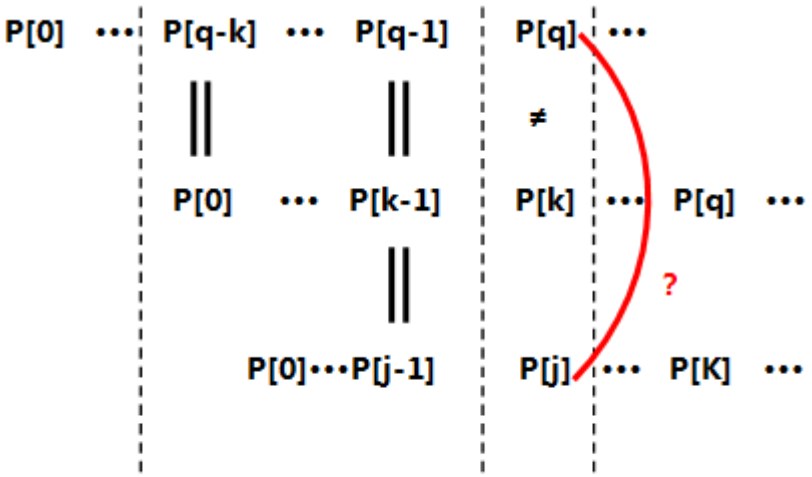


图 2

附代码：

```
#include<stdio.h>
#include<string.h>
void makeNext(const char P[],int next[])
{
    int q,k;
    int m = strlen(P);
    next[0] = 0;
    for (q = 1,k = 0; q < m; ++q)
    {
        while(k > 0 && P[q] != P[k])
            k = next[k-1];
        if (P[q] == P[k])
        {
```

```
14              k++;
15          }
16          next[q] = k;
17      }
18  }
19
20  int kmp(const char T[],const char P[],int next[])
21  {
22      int n,m;
23      int i,q;
24      n = strlen(T);
25      m = strlen(P);
26      makeNext(P,next);
27      for (i = 0,q = 0; i < n; ++i)
28      {
29          while(q > 0 && P[q] != T[i])
30              q = next[q-1];
31          if (P[q] == T[i])
32          {
33              q++;
34          }
35          if (q == m)
36          {
37              printf("Pattern occurs with shift:%d\n",(i-m+1));
38          }
39      }
40  }
41
42  int main()
43  {
44      int i;
45      int next[20]={0};
46      char T[] = "ababxbababcadfdsss";
47      char P[] = "abcdabd";
48      printf("%s\n",T);
49      printf("%s\n",P );
50      // makeNext(P,next);
51      kmp(T,P,next);
52      for (i = 0; i < strlen(P); ++i)
53      {
54          printf("%d ",next[i]);
55      }
56      printf("\n");
57
```

```
58      return 0;
59 }
```