KMP算法

- KMP算法是三位老前辈（D.E.Knuth、J.H.Morris 和 V.R.Pratt）的研究结果，大大的避免重复遍历的情况，全称叫做克努特-莫里斯-普拉特算法，简称KMP算法或看毛片算法。

indexOf()，string.find()的底层实现？

#KMP
经常有一些魔鬼面试官会问你（头条，网易)，你知道indexOf的底层实现吗。。。。？

这就要引申出字符串模式匹配算法，字符串模式匹配算法有很多，比KMP差的也有，比KMP好的也有，那为什么要单独讲KMP这个呢，因为KMP是最经典的字符串匹配算法，很多比kmp优化的算法也是在这个思想基础上修改的，kmp中的前缀后缀，next数组预处理这些思想是最经典的。

前言：字符串模式匹配一般发：暴力匹配

假设现在我们面临这样一个问题：有一个文本串S，和一个模式串P，
现在要查找P在S中的位置，怎么查找呢？

如果用暴力匹配的思路，并假设现在文本串S匹配到 i 位置，
模式串P匹配到 j 位置，则有：

如果当前字符匹配成功（即S[i] == P[j]），则i++，j++，继续匹配下一个字符；

如果失配（即S[i]! = P[j]），令i = i - (j - 1)，j = 0。相当于每次匹配失败时，i 回溯，j 被置为0。

[cpp]

```cpp
int ViolentMatch(char* s, char* p)
{
    int sLen = strlen(s);
    int pLen = strlen(p);

    int i = 0;
    int j = 0;
    while (i < sLen && j < pLen)
    {
        if (s[i] == p[j])
        {
            //①如果当前字符匹配成功（即S[i] == P[j]），则i++，j++
            i++;
            j++;
        }
        else
        {
            //②如果失配（即S[i]! = P[j]），令i = i - (j - 1)，j = 0
            i = i - j + 1;
            j = 0;
        }
    }
    //匹配成功，返回模式串p在文本串s中的位置，否则返回-1
    if (j == pLen)
        return i - j;
    else
        return -1;
}
```

Projects Symbols Files

Workspace
kmp
Sources
main.cpp

```cpp
#include <iostream>

using namespace std;

int main()
{
    int i;
    string tstr = "abcabd";
    string pstr = "abd";
    for(int i = 0; i < 6; i++)
    {
        int flag = 1;
        int cur = i;
        for(int j = 0; j < 3; j++)
        {
            if(tstr[cur]==pstr[j])
                cur++;
            else flag = 0;
        }
        if(flag == 1)
        {
            cout<<i;
            break;
        }

    }
    return 0;
}
```

举个例子，如果给定文本串S“BBC ABCDAB ABCDABCDABDE”，和模式串P“ABCDABD”，现在要拿模式串P去跟文本串S匹配，整个过程如下所示：

1. S[0]为B，P[0]为A，不匹配，执行第②条指令：“如果失配（即S[i]! = P[j]），令i = i - (j - 1)，j = 0”，S[1]跟P[0]匹配，相当于模式串要往右移动一位（i=1，j=0）

```
BBC ABCDAB ABCDABCDABDE

ABCDABD
```

2. S[1]跟P[0]还是不匹配，继续执行第②条指令：“如果失配（即S[i]! = P[j]），令i = i - (j - 1)，j = 0”，S[2]跟P[0]匹配（i=2，j=0），从而模式串不断的向右移动一位（不断的执行“令i = i - (j - 1)，j = 0”，i从2变到4，j一直为0）

```
BBC ABCDAB ABCDABCDABDE

 ABCDABD
```

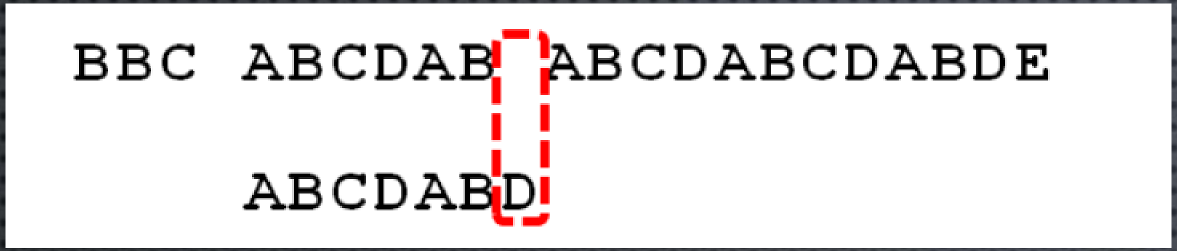
3. 直到S[4]跟P[0]匹配成功（i=4，j=0），此时按照上面的暴力匹配算法的思路，转而执行第①条指令：“如果当前字符匹配成功（即S[i] == P[j]），则i++，j++”，可得S[i]为S[5]，P[j]为P[1]，即接下来S[5]跟P[1]匹配（i=5，j=1）

```
BBC ABCDAB ABCDABCDABDE

    ABCDABD
```

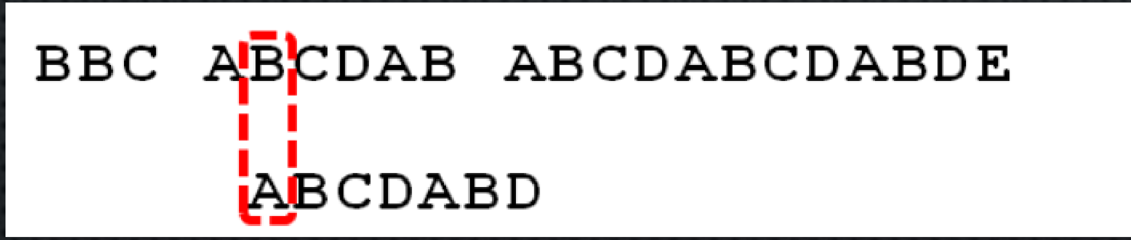
4. S[5]跟P[1]匹配成功，继续执行第①条指令：“如果当前字符匹配成功（即S[i] == P[j]），则i++，j++”，得到S[6]跟P[2]匹配（i=6，j=2），如此进行下去

```
BBC ABCDAB ABCDABCDABDE

    ABCDABD
```

5. 直到S[10]为空格字符，P[6]为字符D（i=10，j=6），因为不匹配，重新执行第②条指令：“如果失配（即S[i]! = P[j]），令i = i - (j - 1)，j = 0”，相当于S[5]跟P[0]匹配（i=5，j=0）



6. 至此，我们可以看到，如果按照暴力匹配算法的思路，尽管之前文本串和模式串已经分别匹配到了S[9]、P[5]，但因为S[10]跟P[6]不匹配，所以文本串回溯到S[5]，模式串回溯到P[0]，从而让S[5]跟P[0]匹配。



```
BBC ABCDAB ABCDABCDABDE
    ABCDABD
```

而S[5]肯定跟P[0]失配。为什么呢？因为在之前第4步匹配中，我们已经得知S[5] = P[1] = B，而P[0] = A，即P[1] != P[0]，故S[5]必定不等于P[0]，所以回溯过去必然会导致失配。那有没有一种算法，让i 不往回退，只需要移动j 即可呢？

答案是肯定的。这种算法就是本文的主旨KMP算法，它利用之前已经部分匹配这个有效信息，保持i 不回溯，通过修改j 的位置，让模式串尽量地移动到有效的位置。

注意红字部分，即……故，说明我们完全有一种办法在模式串里预处理一下，来提前防止(跳过)这种肯定会失败的对比。

KMP算法中的核心部分：next数组，(预处理理得到的，
不占用时间复杂度)

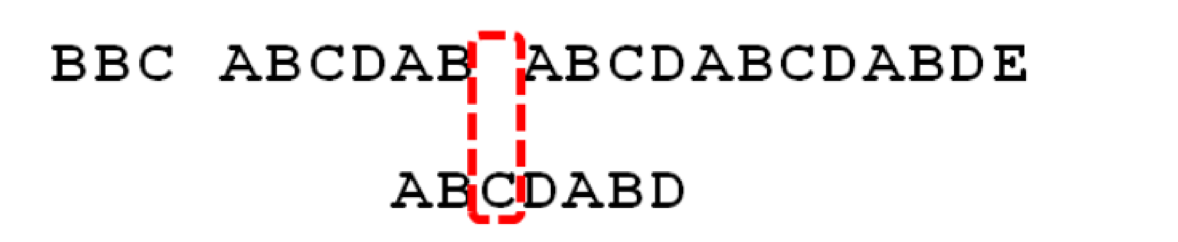
在某个字符失配时，该字符对应的next 值会告诉你下一步匹配中，模式串应该跳到哪个位置（跳到next [j] 的位置）。如果next [j] 等于0或-1，则跳到模式串的开头字符，若next [j] = k 且 k > 0，代表下次匹配跳到j 之前的某个字符，而不是跳到开头，且具体跳过了k 个字符。**即移动的实际位数为：j - next[j]**，且此值大于等于1

还可以省去主串中i的回溯过程，就是主串中指针只扫一遍

这两点正是KMP之所以高效的地方

继续拿之前的例子来说，当S[10]跟P[6]匹配失败时，KMP不是跟暴力匹配那样简单的把模式串右移一位，而是执行第②条指令：“如果j != -1，且当前字符匹配失败（即S[i] != P[j]），则令 i 不变，j = next[j]”，即j 从6变到2（后面我们将求得P[6]，即字符D对应的next 值为2），所以相当于模式串向右移动的位数为j - next[j]（j - next[j] = 6-2 = 4）。

向右移动4位后，S[10]跟P[2]继续匹配。为什么要向右移动4位呢，因为移动4位后，模式串中又有个“AB”可以继续跟S[8]S[9]对应着，从而不用让i 回溯。相当于在除去字符D的模式串子串中寻找相同的前缀和后缀，然后根据前缀后缀求出next 数组，最后基于next 数组进行匹配



```
BBC ABCDAB ABCDABCDABDE

    ABCDABD
```

•①寻找前缀后缀最长公共元素长度对于P = p0 p1 ...pj-1 pj，寻找模式串P中长度最大且相等的前缀和后缀。如果存在p0 p1 ...pk-1 pk = pj- k pj-k+1...pj-1 pj，那么在包含pj的模式串中有最大长度为k+1的相同前缀后缀。举个例子，如果给定的模式串为“abab”，那么它的各个子串的前缀后缀的公共元素的最大长度如下表格所示：

模式串	a	b	a	b
最大前缀后缀公共元素长度	0	0	1	2

比如对于字符串aba来说，它有长度为1的相同前缀后缀a；而对于字符串abab来说，它有长度为2的相同前缀后缀ab

②求next数组

•next 数组考虑的是除当前字符外的最长相同前缀后缀，所以通过第①步骤求得各个前缀后缀的公共元素的最大长度后，只要稍作变形即可：将第①步骤中求得的值整体右移一位，然后初值赋为-1，如下表格所示：

模式串	a	b	a	b
next数组	-1	0	0	1

为什么右移和第一个初始化-1，想想我们匹配的时候，当前是匹配失效的字符，我们找的是失配前字符的最长公共前缀后缀长度。-1是因为第一个都匹配失败的话，-1表示数组下标0的前一个对齐i，表现上即模式串右移。

```
BBC ABCDAB ABCDABCDABDE
        ABCDABD
```

模式串的各个子串	前缀	后缀	最大公共元素长度
A	空	空	0
AB	A	B	0
ABC	A,AB	C,BC	0
ABCD	A,AB,ABC	D,CD,BCD	0
ABCDA	A,AB,ABC,ABCD	A,DA,CDA,BCDA	1
ABCDAB	A,AB,ABC,ABCD,ABCDA	B,AB,DAB,CDAB,BCDAB	2
ABCDABD	A,AB,ABC,ABCD,ABCDA ABCDAB	D,BD,ABD,DABD,CDABD BCDABD	0

这里D失配之后，因为前失配串(不包含当前字符的，最大公共元素长度是2，其实就是右移后的next[6]=2)，即下次匹配的j=next[6]=2，对齐i(文本串的指针),偏移了j-next[6]=6-2=4位。而S串的i是不变的，即不回溯。

字符	A	B	C	D	A	B	D
Next值	-1	0	0	0	0	1	2

```
BBC ABCDAB ABCDABCDABDE
ABCDABD
```

```
BBC ABCDAB ABCDABCDABDE
    ABCDABD
```

```
BBC ABCDAB ABCDABCDABDE
    ABCDABD
```

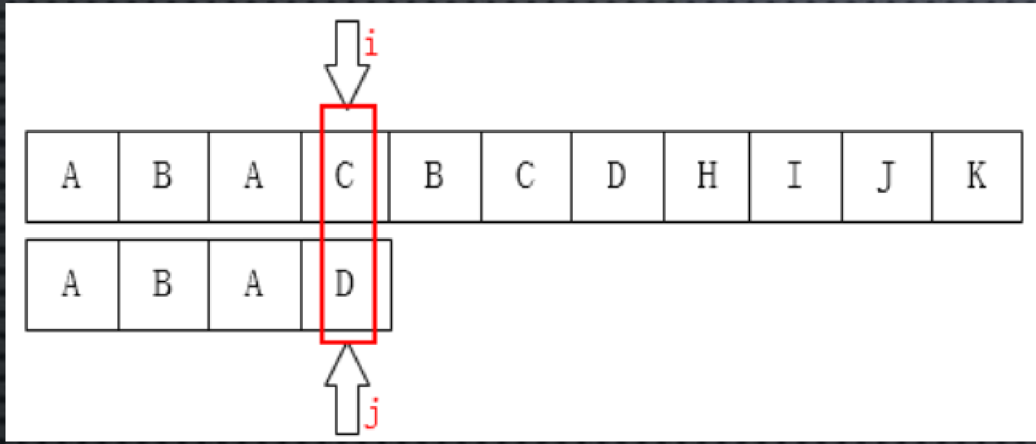
```
BBC ABCDAB ABCDABCDABDE
        ABCDABD
```

```
BBC ABCDAB ABCDABCDABDE
          ABCDABD
```

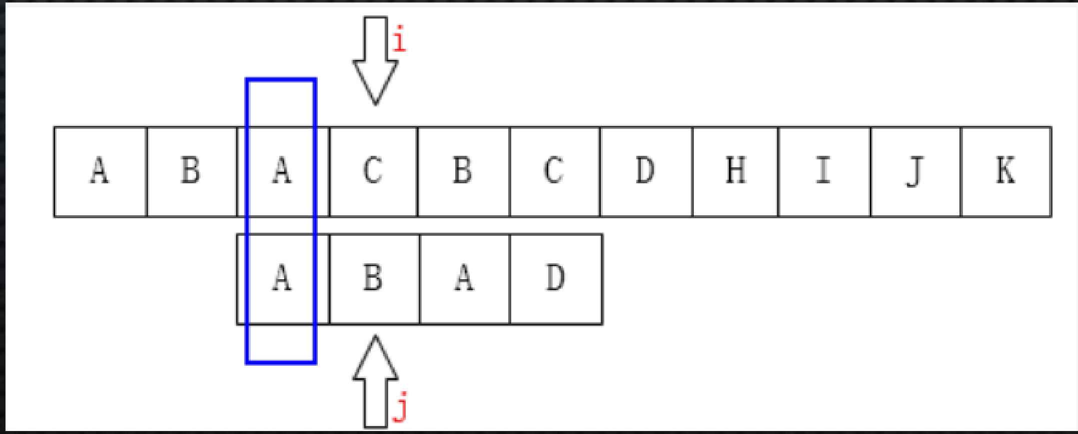
```
BBC ABCDAB ABCDABCDABDE
           ABCDABD
```

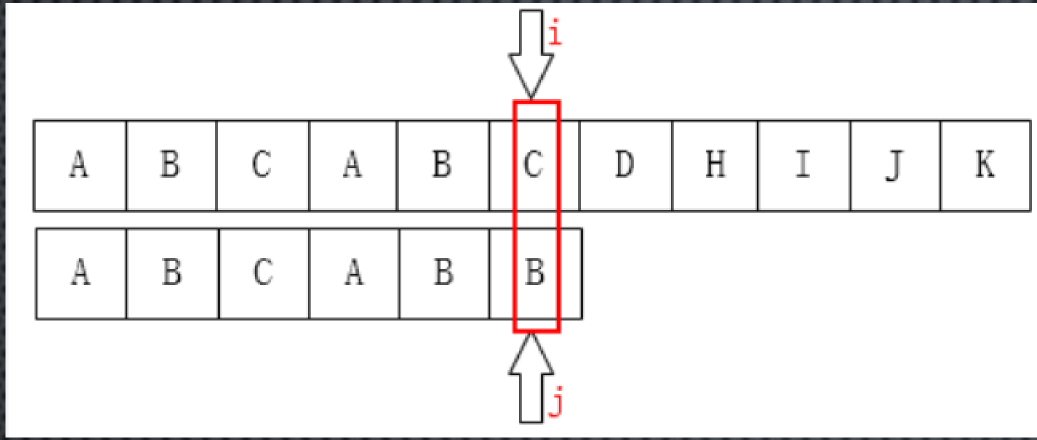
```
BBC ABCDAB ABCDABCDABDE
               ABCDABD
```

所以，整个KMP的重点就在于**当某一个字符与主串不匹配时，我们应该知道j指针要移动到哪**？

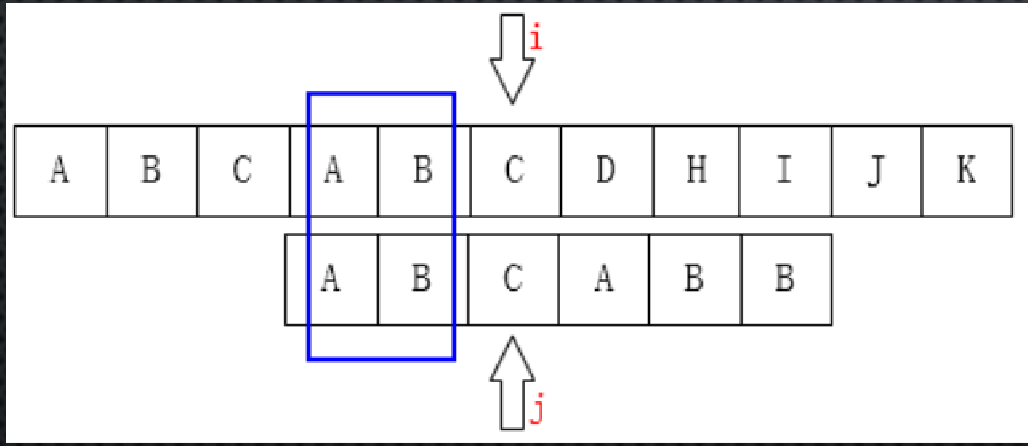


如图：C和D不匹配了，我们要把j移动到哪？显然是第1位。为什么？因为前面有一个A相同啊：





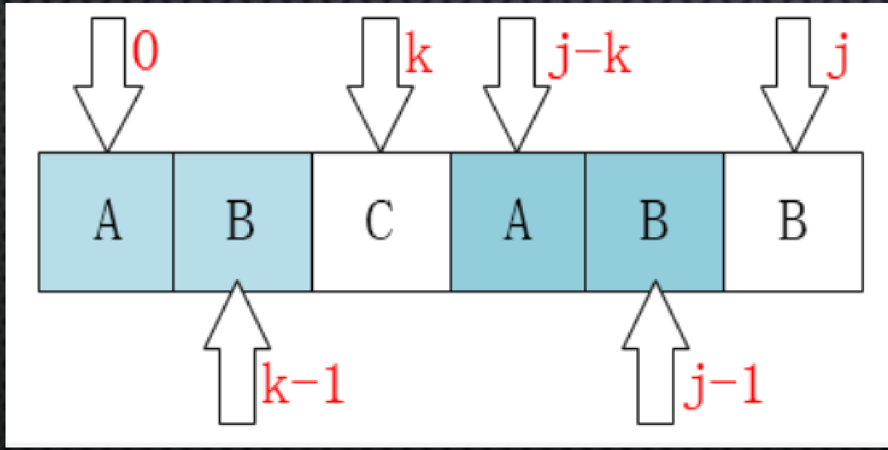
可以把j指针移动到第2位，因为前面有两个字母是一样的：



至此我们可以大概看出一点端倪，当匹配失败时，j要移动的下一个位置k。存在着这样的性质：**最前面的k个字符和j之前的最后k个字符是一样的。**
如果用数学公式来表示是这样的

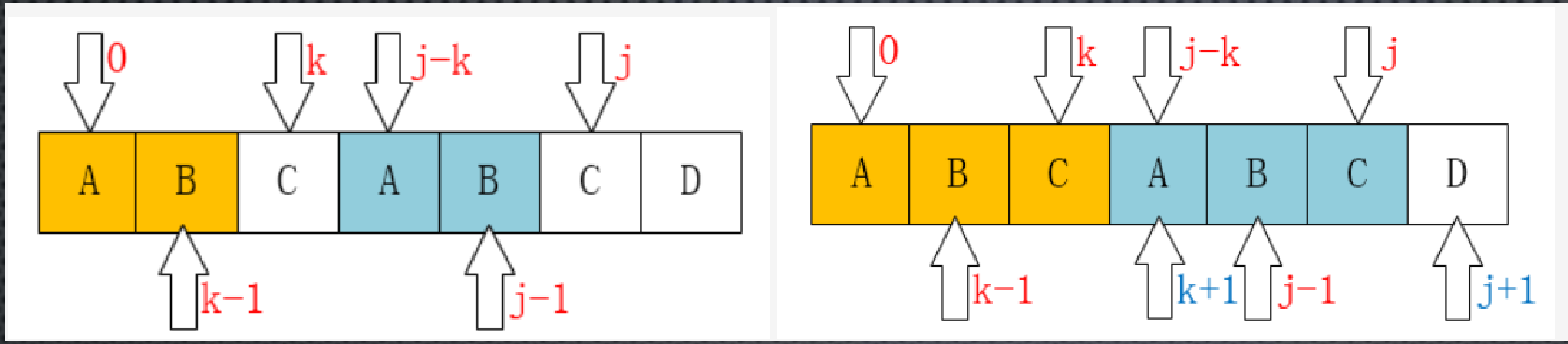
Next[j]=k;

P[0 ~ k-1] == P[j-k ~ j-1]



```c
void GetNext(char* p,int next[])
{
    int pLen = strlen(p);
    next[0] = -1;
    int k = -1;
    int j = 0;
    while (j < pLen - 1)
    {
        //p[k]表示前缀，p[j]表示后缀
        if (k == -1 || p[j] == p[k])
        {
            ++k;
            ++j;
            next[j] = k;
        }
        else
        {
            k = next[k];
        }
    }
}
```

```c
int KmpSearch(char* s, char* p)
{
    int i = 0;
    int j = 0;
    int sLen = strlen(s);
    int pLen = strlen(p);
    while (i < sLen && j < pLen)
    {
        //①如果j = -1，或者当前字符匹配成功（即S[i] == P[j]），都令i++，j++
        if (j == -1 || s[i] == p[j])
        {
            i++;
            j++;
        }
        else
        {
            //②如果j != -1，且当前字符匹配失败（即S[i] != P[j]），则令 i 不变，j = next[j]
            //next[j]即为j所对应的next值
            j = next[j];
        }
    }
    if (j == pLen)
        return i - j;
    else
        return -1;
}
```



请仔细对比这两个图。
我们发现一个规律：
当P[k] == P[j]时，
有next[j+1] == next[j] + 1

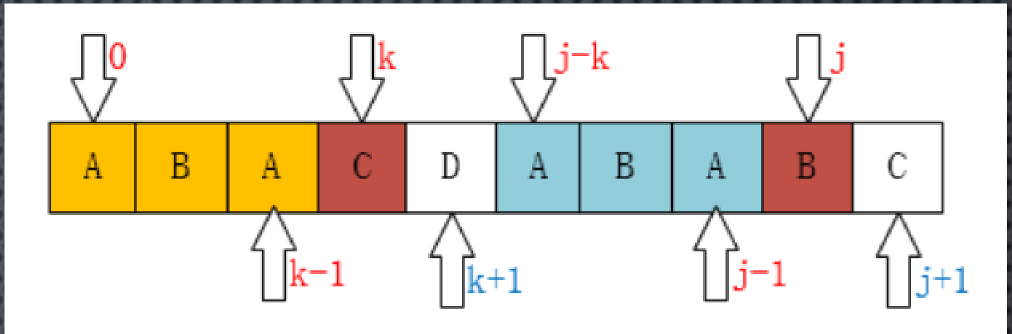
前缀是固定的，后缀是相对的

Next[j]=k

K是前缀，j是后缀

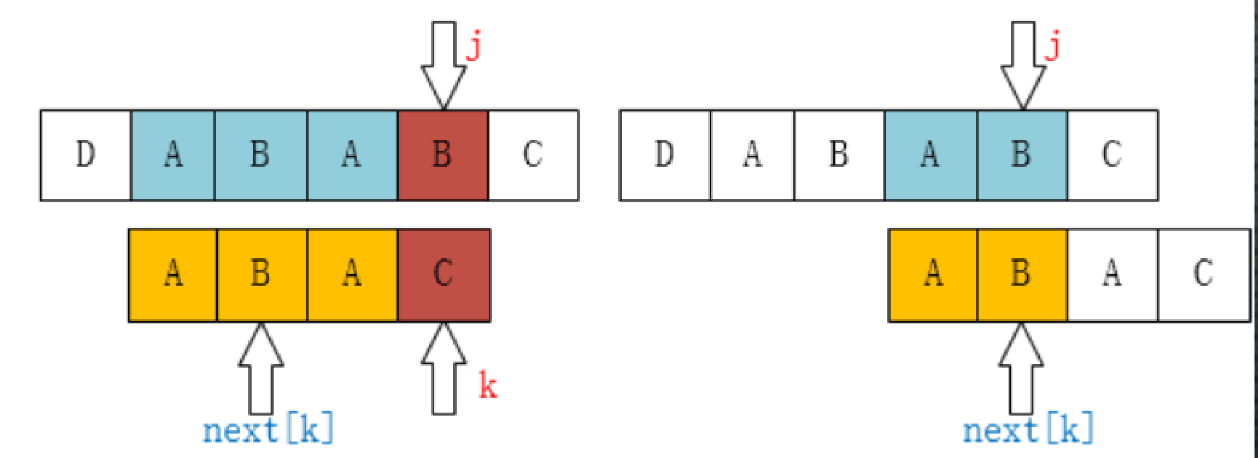
其实这个是可以证明的：
因为在P[j]之前已经有P[0 ~ k-1] == p[j-k ~ j-1]。（next[j] == k）
这时候现有P[k] == P[j]，我们是不是可以得到P[0 ~ k-1] + P[k] == p[j-k ~ j-1] + P[j]。
即：P[0 ~ k] == P[j-k ~ j]，即next[j+1] == k + 1 == next[j] + 1。

那如果P[k] != P[j]呢？比如下图所示：



这个时候next[j+1]直接为0吗

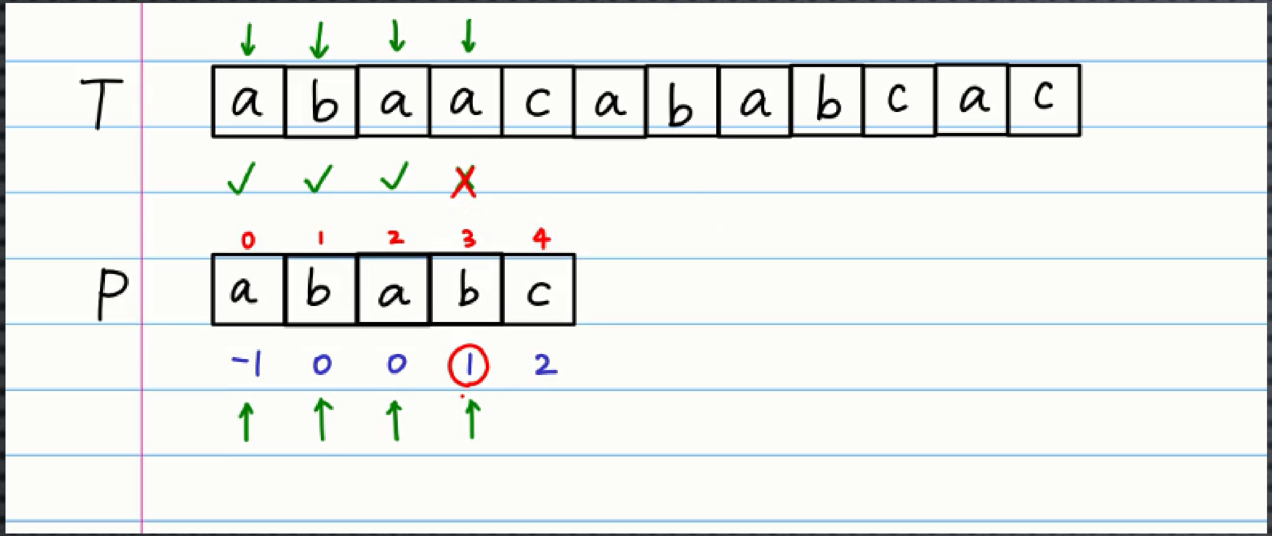
像这种情况，如果你从代码上看应该是这一句：k = next[k];为什么是这样子？



前缀是固定的，后缀是相对的

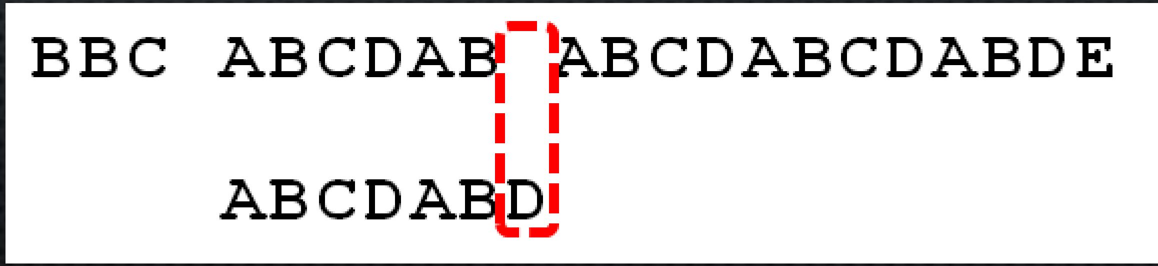
k = next[k]了，像上边的例子，我们已经不可能找到[A，B，A，B]这个最长的后缀串了，但我们还是可能找到[A，B]、[B]这样的前缀串的。所以这个过程像不像在定位[A，B，A，C]这个串，其实就是自我匹配，类似递归的过程。所以next[j+1]=2，而不是直接等于0。当然如果回溯到最后，都没有找到自我匹配的串，那就为0了。

KMP是解决子串和主串关系的一个比较好用的方法，主要是抓住子串的自相似性，也就是在子串中有出现一些相同部分的字符串，然后这些相同部分的子串都被比对成功后，即使在后面匹配出错，也不用重新来，可以利用那一段相同部分做开头继续进行比对




```c
int KmpSearch(char* s, char* p)
{
    int i = 0;
    int j = 0;
    int sLen = strlen(s);
    int pLen = strlen(p);
    while (i < sLen && j < pLen)
    {
        //①如果j = -1，或者当前字符匹配成功（即S[i] == P[j]），都令i++，j++
        if (j == -1 || s[i] == p[j])
        {
            i++;
            j++;
        }
        else
        {
            //②如果j != -1，且当前字符匹配失败（即S[i] != P[j]），则令 i 不变，j = next[j]
            //next[j]即为j所对应的next值
            j = next[j];
        }
    }
    if (j == pLen)
        return i - j;
    else
        return -1;
}
```

继续拿之前的例子来说，当S[10]跟P[6]匹配失败时，KMP不是跟暴力匹配那样简单的把模式串右移一位，而是执行第②条指令：“如果j != -1，且当前字符匹配失败（即S[i] != P[j]），则令 i 不变，j = next[j]”，即j 从6变到2（后面我们将求得P[6]，即字符D对应的next 值为2），所以相当于模式串向右移动的位数为j - next[j]（j - next[j] = 6-2 = 4）。



结束，大家辛苦了，谢谢。