KMP

宁华

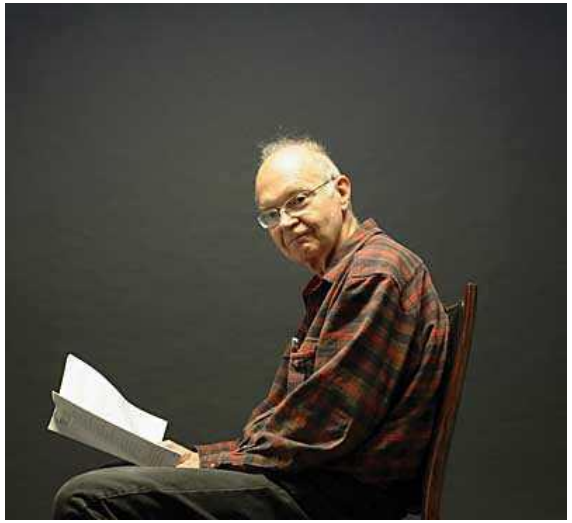
字符串匹配

- 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"？

The Knuth-Morris-Pratt Algorithm

- 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。
- “现代计算机科学的鼻祖”Donald Knuth曾说过“过早的优化是万恶之源”。因为：让正确的程序更快，要比让快速的程序正确容易得多。

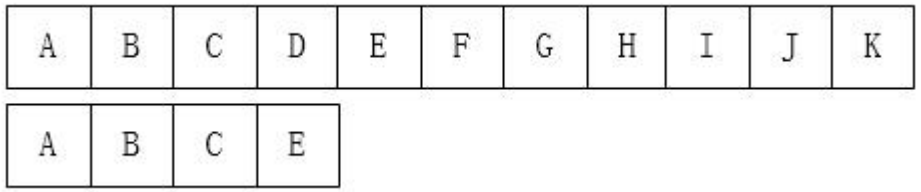


KMP

- KMP： D.E.Knuth、J.H.Morris、 V.R.Pratt。

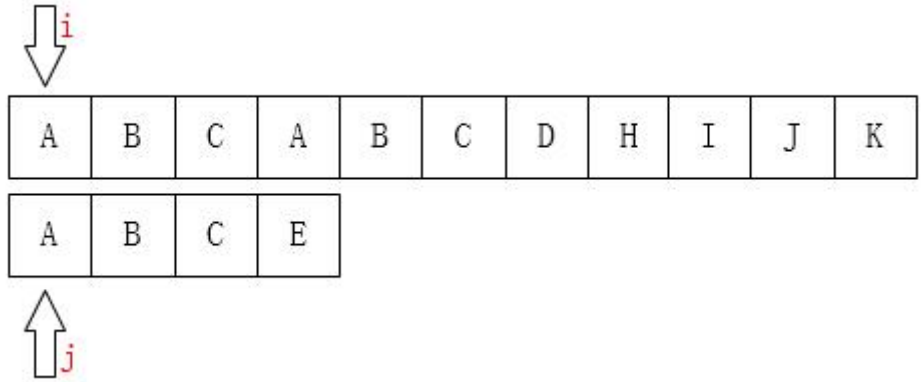
- KMP算法要解决的问题就是在字符串（也叫主串）中的模式（pattern）定位问题。说简单点就是我们平时常说的关键字搜索。模式串就是关键字（接下来称它为P），如果它在一个主串（接下来称为T）中出现，就返回它的具体位置，否则返回-1（常用手段）。

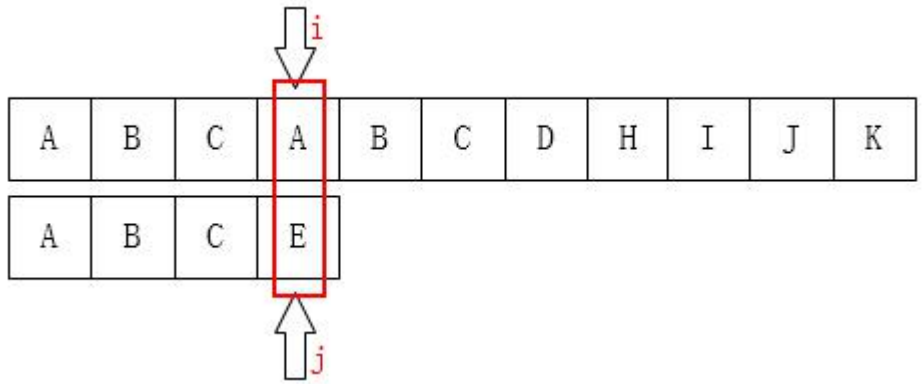
- 主串 T：Text
- 模式串 P： Pattern

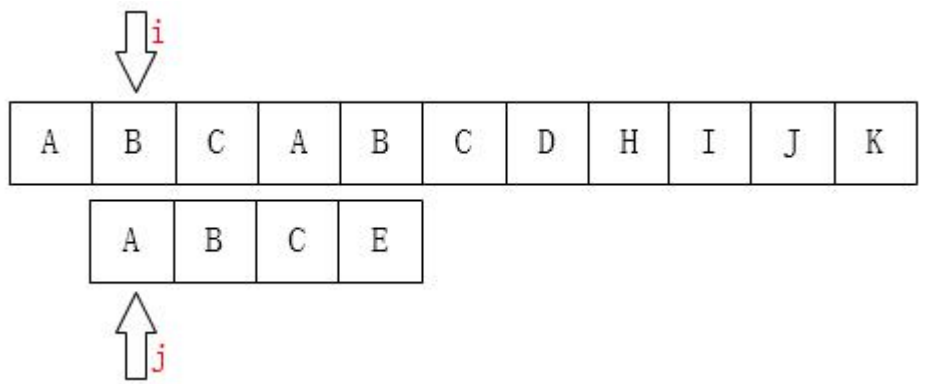


- 注意：之后的讨论，我们认为字符串下标从0开始

- 暴力


i
A B C A B C D H I J K
A B C E
j

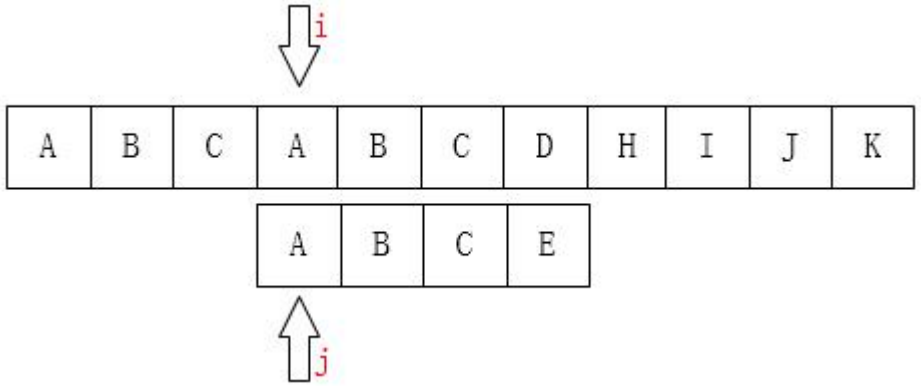

i
A B C A B C D H I J K
A B C E
j


i
A B C A B C D H I J K
A B C E
j

Brute Force

```
int bf(string t, string p)
{
        int i = 0; // 主串的位置
        int j = 0; // 模式串的位置
        while ( i < t.length() && j < p.length() )
        {
                if (t[i] == p[j]) // 当两个字符相同，就比较下一个
                {
                        i++;
                        j++;
                }
                else
                {
                        i = i - j + 1; // 一旦不匹配，i后退
                        j = 0; // j归0
                }
        }
        if (j == p.length()) return i - j;
        else return -1;
}
```

- 如果是人为来寻找的话，肯定不会再把 i 移动回第1位，因为**主串匹配失败的位置前面除了第一个A之外再也没有A了**，我们为什么能知道主串前面只有一个A？因为我们已经**知道前面三个字符都是匹配的！（这很重要）**。移动过去肯定也是不匹配的！有一个想法，**i 可以不动，我们只需要移动 j 即可**，如下图：

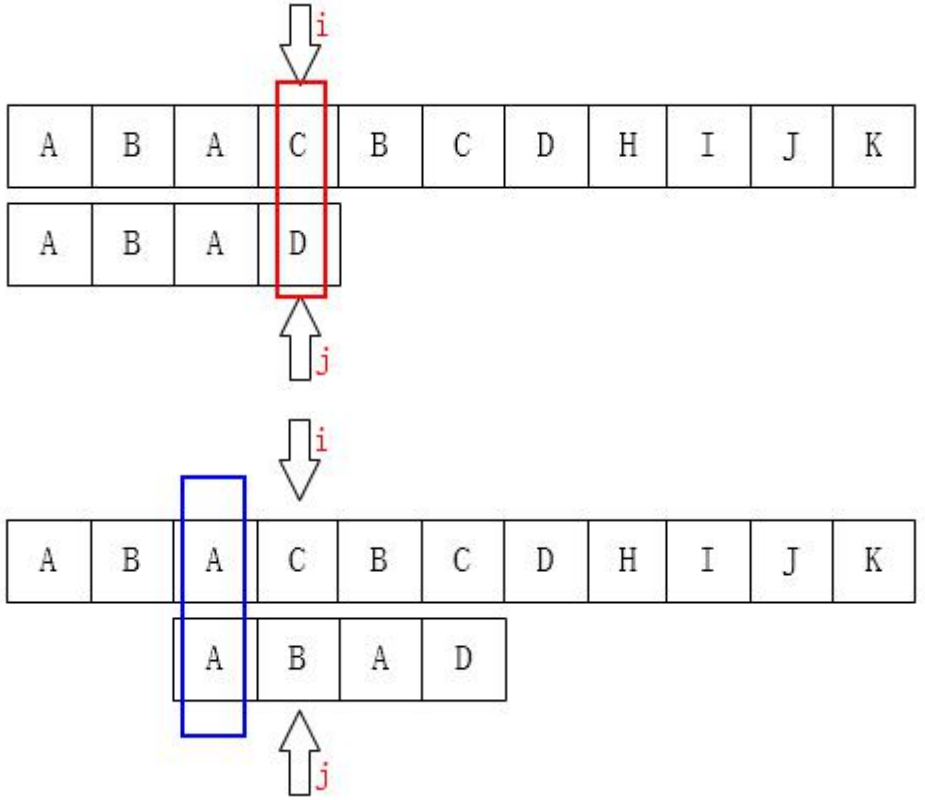


- 大牛们是无法忍受“暴力破解”这种低效的手段的，于是他们三个研究出了KMP算法。其思想就如同我们上边所看到的一样：“**利用已经部分匹配这个有效信息，保持 i 指针不回溯，通过修改 j 指针，让模式串尽量地移动到有效的位置。**”

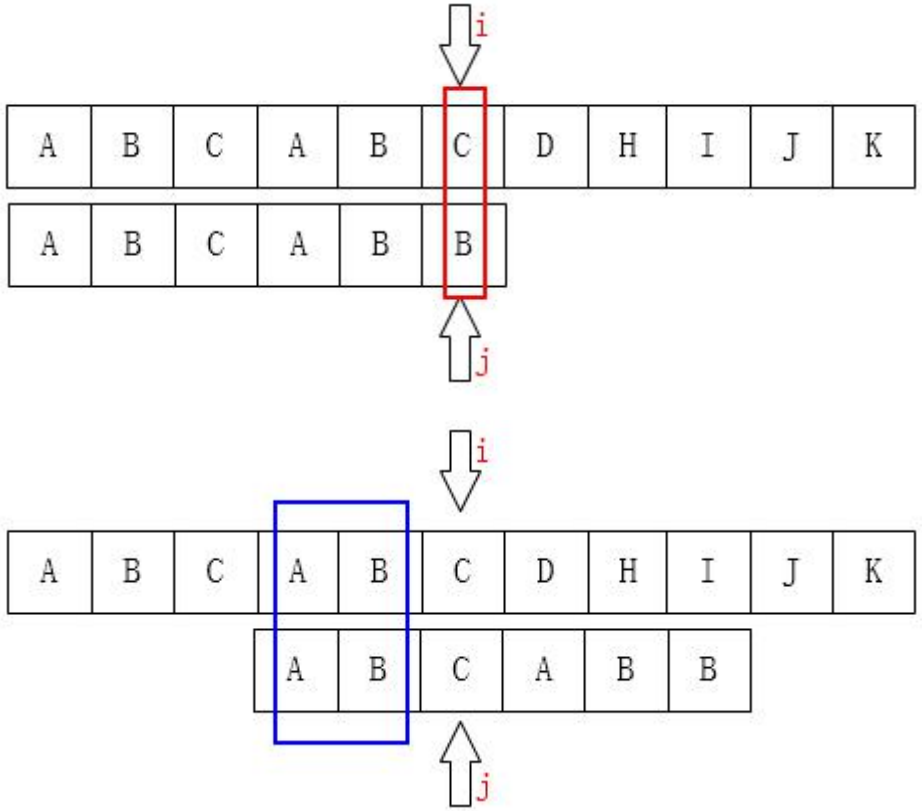
- 所以，整个KMP的重点就在于当某一个字符与主串不匹配时，我们应该知道 j 指针要移动到哪？

- 接下来我们自己来发现j的移动规律：

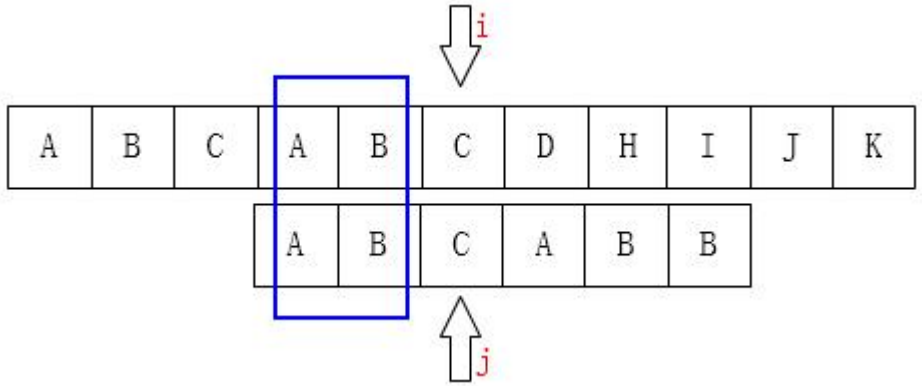
- 如图：C和D不匹配了，我们要把j移动到哪？显然是第1位(下标从0开始)。为什么？因为前面有一个A相同啊：



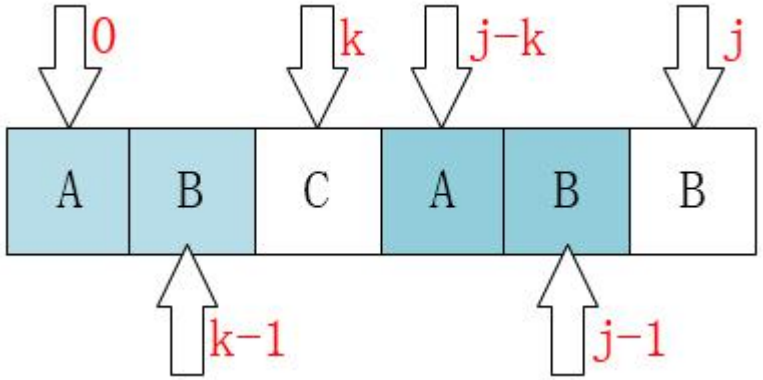
- 如下图也是一样的情况：



- 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。

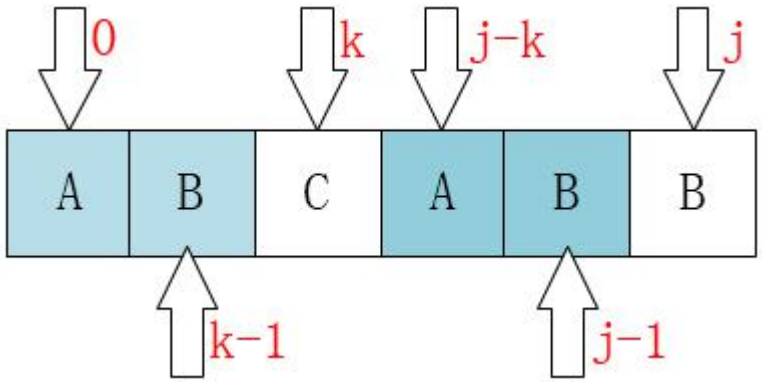


- 如果用数学公式来表示是这样的
- P[0 ~ k-1] == P[j-k ~ j-1]



- 弄明白了这个就应该可能明白为什么可以直接将 j 移动到位置 k 了。

- 因为:
- 当T[i] != P[j]时
- 有T[i-j ~ i-1] == P[0 ~ j-1]
- 由P[0 ~ k-1] == P[j-k ~ j-1]
- 必然：T[i-k ~ i-1] == P[0 ~ k-1]



- 接下来就是重点了，怎么求这个（这些）k 呢？因为在P的每一个位置都可能发生不匹配，也就是说我们要计算每一个位置 j 对应的 k，所以用一个数组 nxt 来保存，nxt[j] = k，表示当 T[i] != P[j] 时，j 指针的下一个位置。

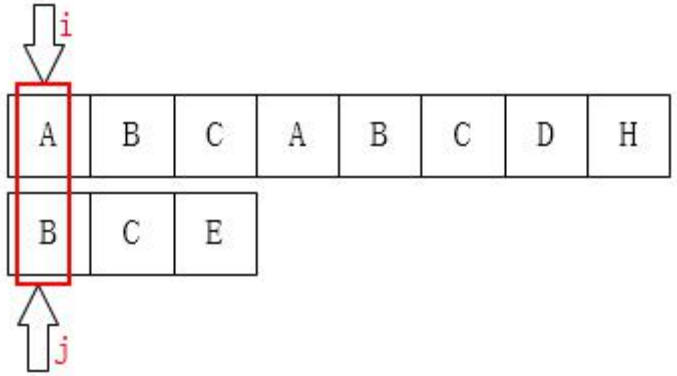
- 接下来的任务：求出 nxt 数组

- 暴力?

求nxt数组

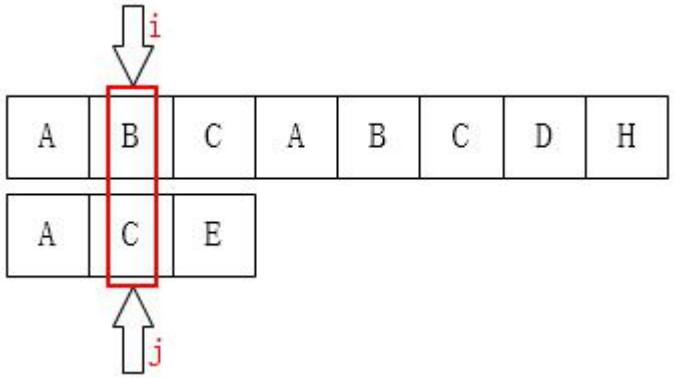
```
void getNext(string p)
{
        nxt[0] = -1;
        int j = 0;
        int k = -1;
        while (j < p.length() - 1)
        {
                if (k == -1 || p[j] == p[k])
                        nxt[++j] = ++k;
                else
                        k = nxt[k];
        }
}
```

- 先来看第一个：当j为0时，如果这时候不匹配，怎么办？



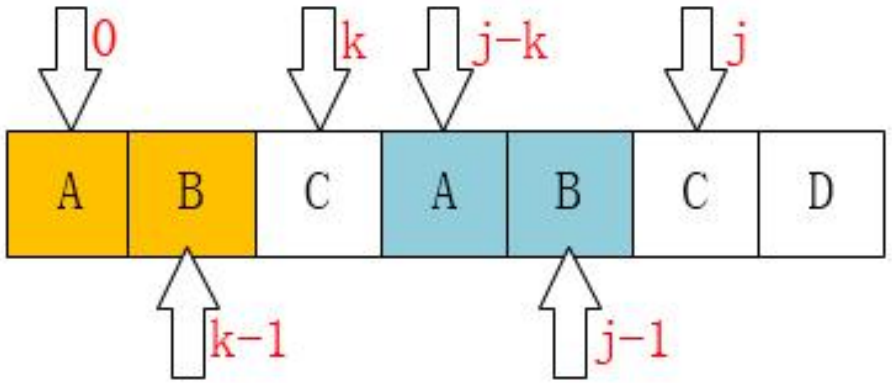
- 像上图这种情况，j已经在最左边了，不可能再移动了，这时候要应该是i指针后移。所以在代码中才会有 nxt[0] = -1; 这个初始化。

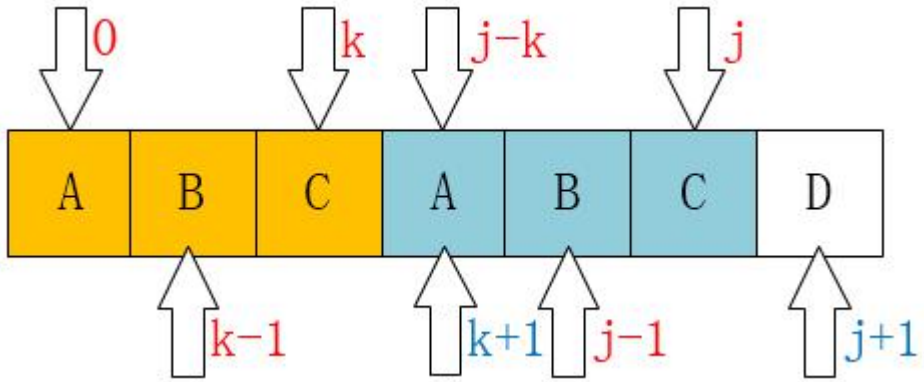
- 如果是当j为1的时候呢?



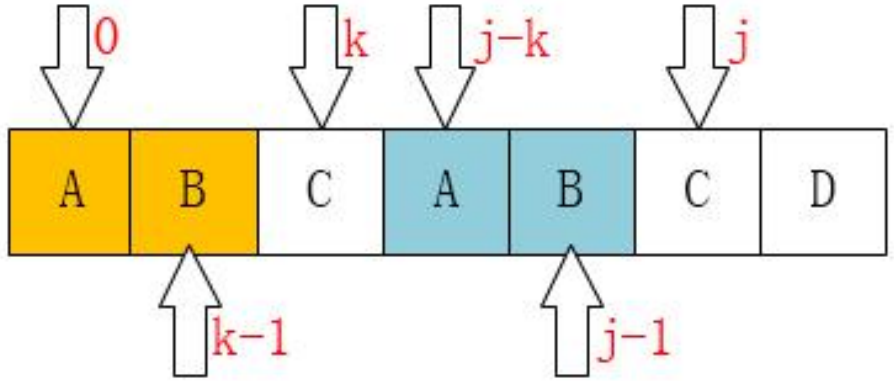
- 显然，j指针一定是移到0位置的。因为它前面也就只有这一个位置了 nxt[1] = 0 ;

- 下面这个是最重要的，请看如下图：



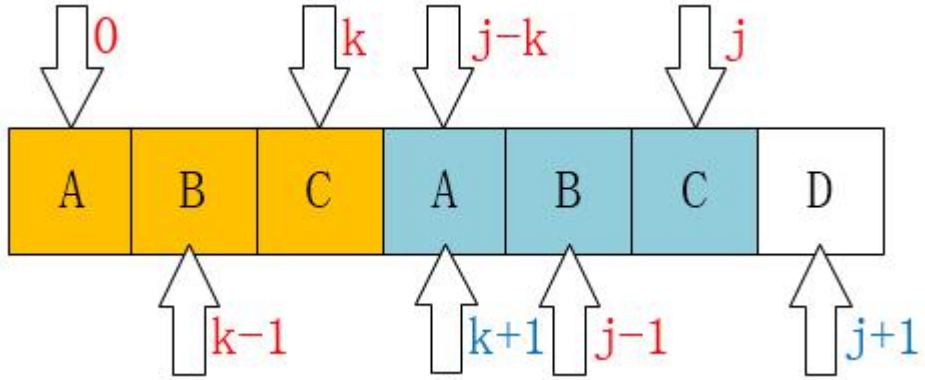


- 请仔细对比这两个图。



- 我们发现一个规律：

- 当P[k] == P[j]时，有nxt[j+1] == nxt[j] + 1

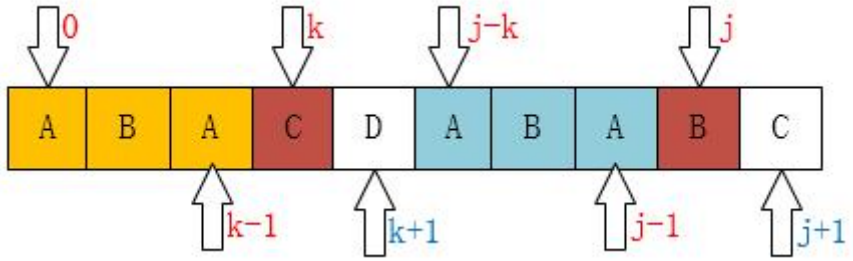


- 其实这个是可以证明的：

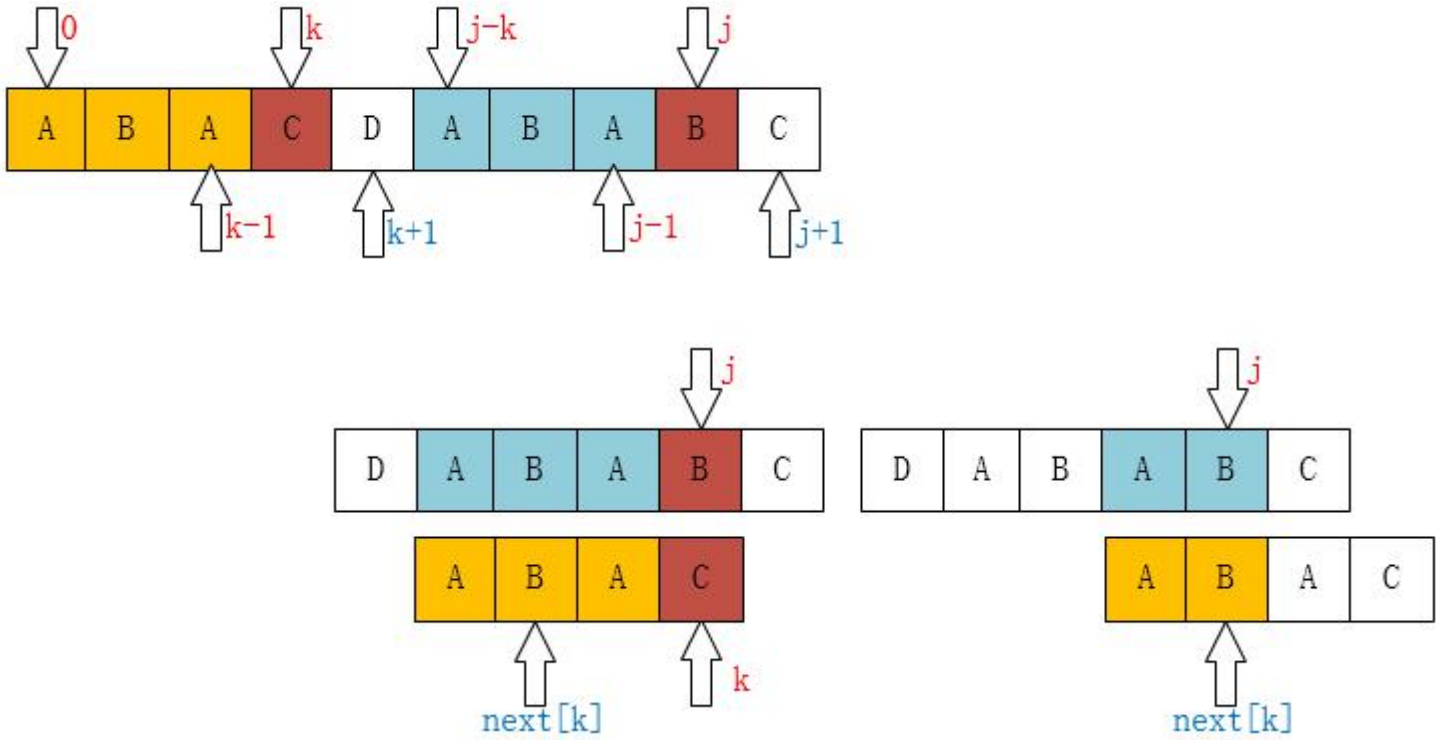
- 因为在P[j]之前已经有P[0 ~ k-1] == p[j-k ~ j-1]。 （nxt[j] == k）
- 这时候现有P[k] == P[j]，我们是不是可以得到P[0 ~ k-1] + P[k] == p[j-k ~ j-1] + P[j]。

- 即： P[0 ~ k] == P[j-k ~ j]，即nxt[j+1] == k + 1 == nxt[j] + 1。

- 那如果P[k] != P[j]呢？比如下图所示：



- 像这种情况，如果你从代码上看应该是这一句： k = nxt[k];为什么是这样子？你看下面应该就明白了。



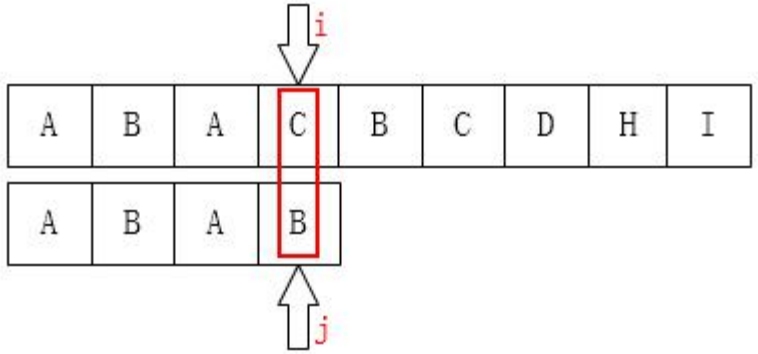
- 现在你应该知道为什么要k = nxt[k]了吧！像上边的例子，我们已经不可能找到[A，B，A，B]这个最长的后缀串了，但我们还是可能找到[A，B]、[B]这样的后缀串的。所以这个过程像不像在定位[A，B，A，C]这个串，当C和主串不一样了（也就是k位置不一样了），那当然是把指针移动到next[k]啦。

比较暴力算法和KMP算法代码

```
int bf(string t, string p)
{
    int i = 0; // 主串的位置
    int j = 0; // 模式串的位置
    int lent = t.length();
    int lenp = p.length();
    while ( i<lent && j<lenp )
    {
        if (t[i] == p[j])
        {
            i++;
            j++;
        }
        else
        {
            i = i - j + 1; // i 后退
            j = 0; // j 归 0
        }
    }
    if(j==lenp)return i-lenp;
    else return -1;
}
```

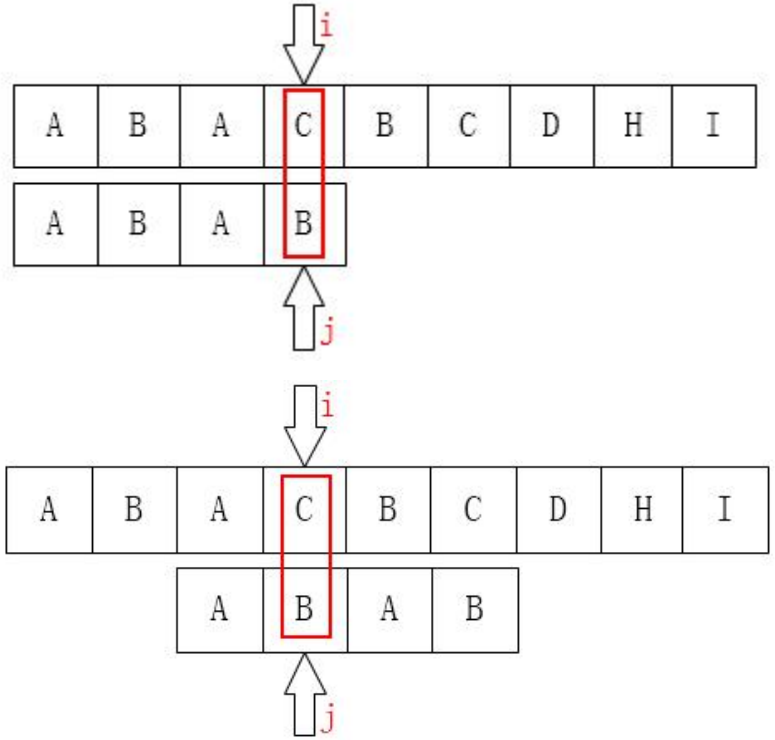
```
int KMP(string t, string p)
{
    int i = 0; // 主串的位置
    int j = 0; // 模式串的位置
    int lent = t.length();
    int lenp = p.length();
    while ( i<lent && j<lenp )
    {
        if (j==-1 || t[i] == p[j])
        {
            i++;
            j++;
        }
        else
        {
            //i = i - j + 1; // i 不回溯了
            j = nxt[j];
        }
    }
    if(j==lenp)return i-lenp;
    else return -1;
}
```

- 最后，来看一下上边的算法存在的缺陷。来看第一个例子：



- 显然，当我们上边的算法得到的next数组应该是[-1，0，0，1]

- 所以下一步我们应该是把j移动到第1个元素咯：



- 不难发现，这一步是完全没有意义的。因为后面的B已经不匹配了，那前面的B也一定是不匹配的，同样的情况其实还发生在第2个A元素上。

- 显然，发生问题的原因在于P[j] == P[next[j]]。

- 所以我们也只需要添加一个判断条件即可：

修改

```
void getNext(string p)
{
        nxt[0] = -1;
        int j = 0;
        int k = -1;
        while (j < p.length() - 1)
        {
                if (k == -1 || p[j] == p[k])
                        nxt[++j] = ++k;
                else
                        k = nxt[k];
        }
}
```

```
void getNext(string p)
{
        nxt[0] = -1;
        int j = 0;
        int k = -1;
        while (j < p.length() - 1)
        {
                if (k == -1 || p[j] == p[k])
                {
                        if (p[++j] == p[++k])
                                next[j] = next[k];
                        else next[j] = k;
                }
                else k = nxt[k];
        }
}
```