# 算法基础（六）：KMP算法详解

# KMP算法

## 几个最基本的概念：

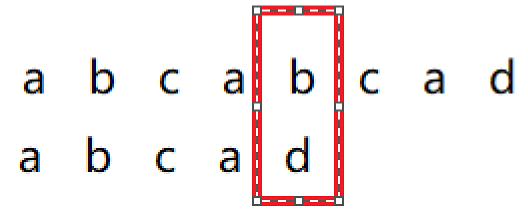
1. 字符串的**前缀**：从主串下标0开始的子串称为主串的前缀
2. 字符串的**后缀**：从主串下标大于0的位置到结尾的子串称为主串的后缀
3. **目标串**：也就是主串，简单说就是那条比较长的串
4. **模式串**：也就是那条短的，用来匹配的串
5. kmp算法的**目的**：在`O(m+n)`的时间复杂度的内进行串匹配，也就是在目标串中找到模式串，并返回目标串中模式串的第一个字符下标

要了解基本思想之前需要先看看暴力算法匹配字符串怎么做。

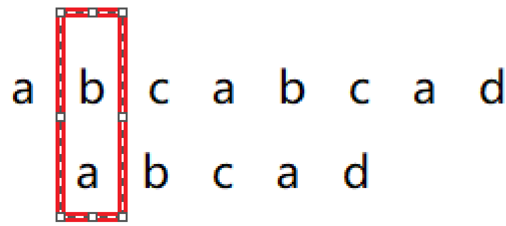
## 暴力算法：

以`s[] = a b c a b c a d`与`p[] = a b c a d`为例

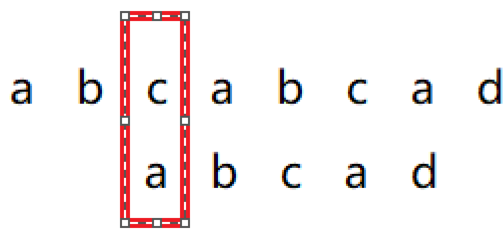
**第一次**：`s[0]`开始匹配`s[4] != p[4]`，不匹配



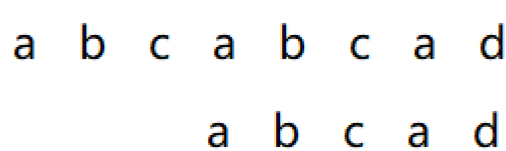

**第二次：**`p[]`向后移动一次，从`s[1]`开始匹配，直接`s[1] != p[0]`，不匹配



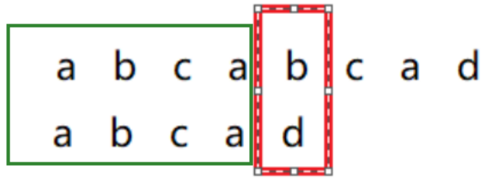

**第三次**：`p[]`向后移动一次，从`s[2]`开始匹配，发现又是直接不匹配




**第四次：**`p[]`再向后移动一次，匹配成功



我们发现，暴力算法每次匹配失败后都是在目标串`s[]`中向后移动一个字符，然后开始匹配，说明暴力算法并没有吸取前面匹配失败的经验，每次都是从头开始，这里的经验值得是什么呢，以上面的匹配为例，经验指的就是：**在第一次匹配失败后，前面四个字符是匹配成功的这个信息**，如下面的绿色框所示：



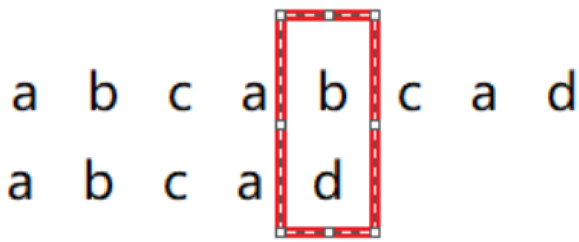

所以KMP算法的思想可以总结为：**利用匹配失败的相关信息来进行某些操作，吸取教训从而减少暴力算法的尝试次数（跳过某些尝试），从而降低时间复杂度。**下面来说明kmp算法的步骤。

# KMP算法：

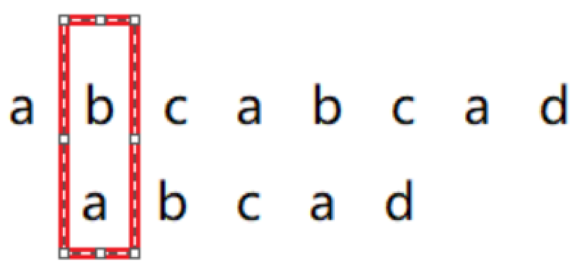
## 基本思想

要彻底理解这个算法的思想，我们还得分析暴力算法的过程，上文说到，这个算法的思想就是减少暴力算法的尝试次数，那么我们来分析这些跳过的次数。

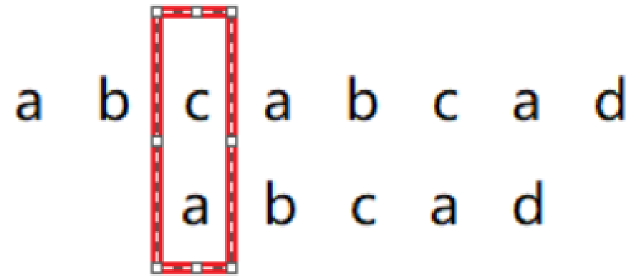
第一次，`b`和`d`不匹配，目标串后移一个字符，重新开始匹配。



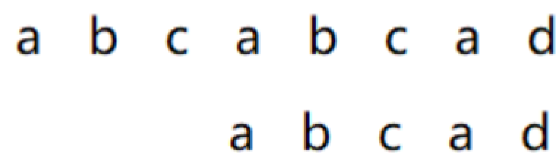

第二次，目标串后移一次后我们可以看到，目标串中的`s[1..3] = b c a`与模式串中的`p[0..2] = a b c`是不同的，同时由于第一次的匹配信息，目标串中的`s[1...3] = p[1...3]`，也就可以说，模式串的`p[0...2] != p[1..3]`，在`p[0..3]`这个串中，最长的前缀不等于后缀，这一次尝试在kmp中需要跳过。




第三次，目标串再次后移一位，目标串中的`c a != a b`也就等价于模式串中`p[0..1] != p[2..3]`，在`p[0...3]`这个串中，次长的前缀不等于后缀，跳过。




第四次，目标串再次后移一位，这一次也是kmp算法直接跳到的位置，我们可以看到，这一次`a = a`也就是在模式串中`p[0] = p[3]`最短的一个前缀等于后缀，这时算法不再跳过。




所以我们看到，由于第一次匹配记录的信息，kmp算法跳过的那几个暴力尝试，**这些失败匹配都有一个特征：模式串中`p[0...i]`的 前缀与后缀不匹配，所以我们可以说，只要前缀与后缀不同的尝试，我们都需要跳过。**

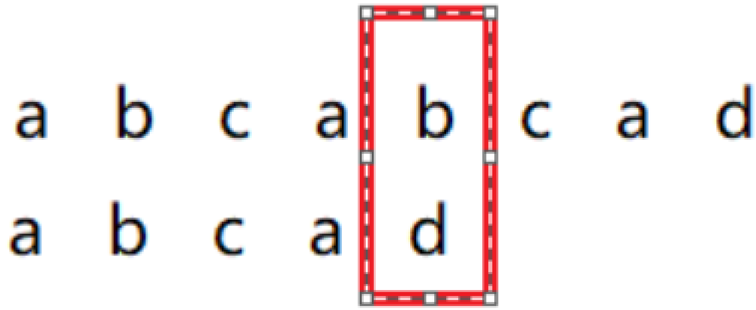
所以kmp算法思想的更进一步表达就是：**在一次整体匹配失败后我们必定可以得到一些匹配成功的串，我们发现在后面的匹配尝试中，这些匹配成功的串只要出现后缀不等于前缀的情况，那这些尝试就必定是失败的，于是我们可以直接跳过这些尝试，直接进行后缀等于前缀的尝试，至于这个尝试是不是失败我们根据经验是不知道的，我们接着递归这个过程，直到匹配完全。**

## 数组`next[i]`的含义：

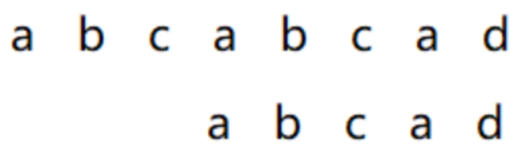
`next[i] = k`表示`p[0....i]`这个串中，前缀与后缀相同的情况下，前缀的最长长度为`k`，比如在上面的例子中`next[3] = 1`是因为`p[0...3] = a b c a`前缀与后缀相等也就是`a = a`长度也就是`1`，或者再举一个例子：`a c d e f a c d e`，这里`next[6] = 2(a c = a c), next[8] = 4(a c d e = a c d e)`

## 基本操作：

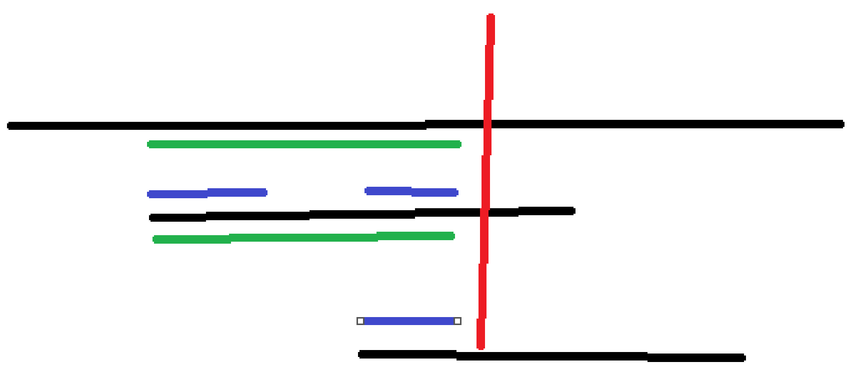
当有一个元素不匹配的时候，我们将模式串`p[]`进行后移，直接跳过注定不匹配的尝试，进行后面的操作，以上面为例：




跳过第二次，第三次，直接进行第四次匹配：




怎么做到呢，通过`next[]`数组，当进行第一次匹配失败时，我们发现`p[]`中，`next[3] = 1`于是，直接将`p[]`向后移动`3 - 1 + 1`个位置对齐，也就是移动`i - k + 1`个位置，如下图：



绿色表示目标串与模式串匹配的部分，红色表示这个字符不匹配，蓝色表示模式串中前缀与后缀相同的最长长度，当不匹配的时候，就将模式串向后移动，一直移动到前缀与后缀相同的位置，也就是移动`i - k + 1`个位置，跳过必然不可能的尝试，从这个时候再次进行尝试，一直到匹配成功。

## 几个问题：

1. 为什么，分析前缀后缀的时候是用模式串中的`p[0..i]（本例中是p[0..3]）`部分？
    1. 我们第一次匹配得到的经验就是`s[0..3] = p[0...3]`，在后续使用这条经验跳过的尝试始终来自`s[0..3]（s中a开头的尝试，b开头的尝试，c开头的尝试，a开头的尝试）`，由上面对比字符的过程我们也可以看到，我们始终是用`s[0...3]`的后缀部分与`p[0..3]`的前缀部分进行对比，从而跳过尝试，所以直接分析前缀后缀的时候必须限定在`p[0...3]`

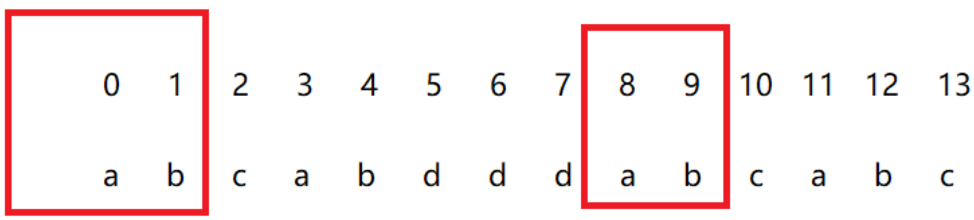
2. 为什么要求`next[i] = k`这里的`k`指的是最长的前缀和后缀呢？
   1. 同样由上面的暴力分析过程，尝试的过程是一个字符一个字符往后移的，当满足后缀等于前缀的时候就不再跳过，此时的后缀前缀显然是最长的一个。
3. 怎么实现`next[]`数组呢？
   1. 其实`next[]`数组的实现才是kmp算法最难最核心的东西，下文来实现`next[]`数组

## 第一种`next[]`数组的快速实现：

上文中已经说明了`next[i] = k`的含义，而且`k != i + 1`否则自己与自己相等也就没有意义了。

首先，我们如果采用暴力做法的话，与前面的匹配规则类似，也是一个一个比较，不同的是自己与自己比较，一个个往后移，时间复杂度是`O(m^2)`，这样的话，我们之前的优化就没有意义了，所以这里我们必须进一步优化。我们以字符`p[] = a b c a b d d d a b c a b c`为例来说明这个实现过程。

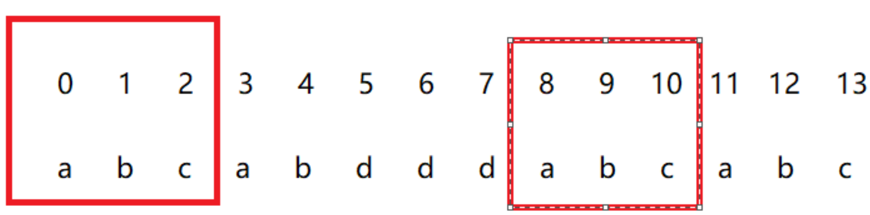
在求解过程中假设我们已经求得了`next[x-1] = now`也就是说以`p[x-1]`结尾的字符串，其前缀与后缀相同的最长长度是`now`如下图，`next[9] = 2 = now`：



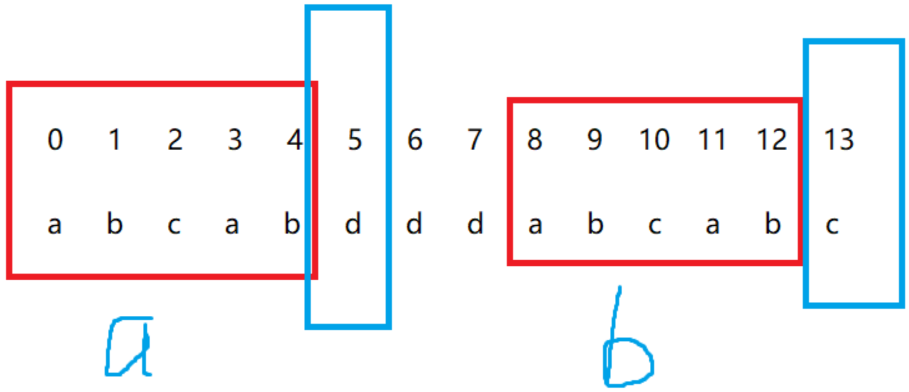

那么对于`p[x]`则有两种情况：

1. `p[x] = p[now]`如上图的举例
2. `p[x] != p[now]`

对于第一种情况比较简单，因为以`p[x-1]`结尾的字符串，前缀与后缀相等，最长的长度是`now`，而`p[x]` `(p[10])`又等于`p[now](p[2])`那以`p[x]`结尾的字符串的前缀与后缀相等的最大长度，直接就是`now + 1`了，扩充一下即可，如下图：




对于第二种情况，`p[x] != p[now]`比如出现了`p[13] != p[5]`如下图，此时`now = 5`

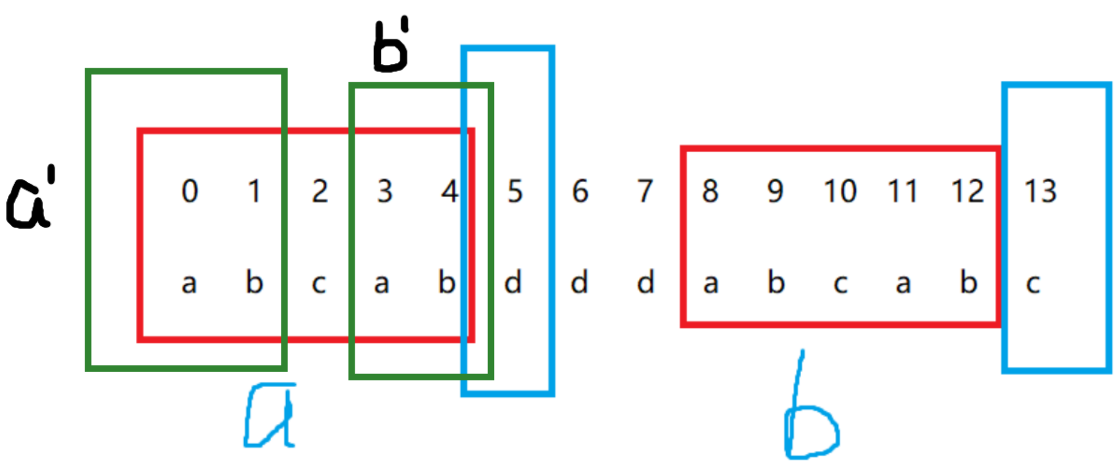



在这时，我们肯定不能直接在原来的`now`上直接加`1`对吧，但是我们要明白一个事实：那就是以`p[x]`这个字符结尾的串，其前缀与后缀相等时，最大的长度假设为`now'`，那么，将这个前缀和后缀去掉一个字符`p[x]`后，得到的两个新的串也必然是相等的，而且也分别是以`p[x-1]`结尾的串的前缀和后缀，同时也是`a`的前缀和`b`的后缀，而且这两个新串的长度必然小于`now`(因为`now`是以`p[x-1]`结尾的最大前缀与后缀相等情况下的长度)。以上面为例，以`p[13]`结尾的串前缀与后缀相等时，最大的长度我们用肉眼分析一下是`a b c`，那么去掉`c`之后，显然`a b`也是以`p[12]`结尾的前缀和后缀，也是`a`的前缀和`b`的后缀，并且小于`now = 5`。

所以，第二种情况就可以转化为：我们需要找到以`p[x-1]`结尾的串的前缀和后缀相等情况下第二长的长度`now1`看`p[x]`是否等于`p[now1]`，若相等，则`now' = now1 + 1`，若不相等，则递归看第三长的长度`now2`，看是否有`p[now2] = p[x]`如此这样递归下去，**总之，就是找`a`的前缀与`b`的后缀相等情况下尽可能大的长度，来进行对比。**

在找第二长的长度时，我们可以发现，以`p[x-1]`结尾的串的前缀和后缀相等情况下第二长的长度，恰好是`a`的前缀与`b`的后缀相等时的最长长度，而由于`a`与`b`相等，这最长长度也就等价于`a`串的前缀与后缀相等时的最长长度，恰好就是`next[4] = next[now - 1] = 2 = now1`，这里也就是`a b`

当第二长长度不满足`p[x] = p[now1]`的时候，我们需要找以`p[x-1]`结尾的串的前缀和后缀相等情况下第三长的长度，而这第三长的长度也就是`a`的前缀与`b`的后缀相等情况下第二长的长度，也就是`a`串前缀与后缀相等时的第二长长度，与找以`p[x-1]`结尾的串的前缀和后缀相等情况下第二长的长度的思想类似，我们通过递归分析，`a`串前缀与后缀相等时的第二长长度，也就是`a'`串的前缀和`b'`串的后缀相等时的最长长度，，如下图绿色框框，也就是`a'`串的前缀与后缀相等时的最长长度：




**我们再将上面的一堆文字用数学语言规范一下**

**定义：**

1. `next[x]` 表示以字符 `p[x]` 结尾的串前缀与后缀相同情况下的最长长度
2. `n[x]` 表示以字符 `p[x]` 结尾的串前缀与后缀相同情况下的长度
3. `now` 表示 `next[x-1]` 的值，也就是以 `p[x-1]` 结尾的串前缀与后缀相等情况下的最大长度，作为下标的时候也就是上文中左边蓝色框出的字符

**注意到：**

无论 `p[x]` 与 `p[now]` 是否相等，都有等式：`next[x] = n[x-1] + 1` 成立，且 `n[x-1]` 的取值是 `0 ~ next[x-1]`（解释在上文，不再赘述）

**要求：**

`next[x]` 最大，那必然 `n[x-1]` 最大，所以接下来分两种情况来讨论 `n[x-1]` 最大的的取值。

1. 当 `p[x] = p[now]` 的时候，显然 `n[x-1]` 取 `next[x-1]` 即可，前缀和后缀可以连起来
2. 当 `p[x] != p[now]` 的时候，`n[x-1]` 不能取 `next[x-1]`，那显然 `n[x-1]` 得取一个次大的值（小于 `now`），再由于此时 `n[x-1]` 的定义，可以得出：
    1. 以 `p[x-1]` 结尾的串的前缀与后缀同时也是上图中 `a` 的前缀和 `b` 的后缀
    2. 由于 `a` 串与 `b` 串相等，所以 `b` 的后缀也是 `a` 的前缀，所以以 `p[x-1]` 结尾的串的前缀与后缀同时也是上图中 `a` 的前缀和后缀
    3. 所以在小于 `now` 的情况下，求 `n[x-1]` 的最大值，也就是 `a` 串前缀与后缀相等情况下的最长长度
    4. 所以我们此时将 `now` 递归为 `a` 串前缀与后缀相等情况下的最长长度，即 `now = next[now-1]`

递归过后我们得到了 `n[x-1]` 的次长长度，这时仍需要比较 `p[x]` 与 `p[now]`，看是否有等式 `next[x] = n[x-1] + 1` 成立，否则，继续递归，一直到 `n[x-1] = 0` 的时候，这时表示以 `p[x-1]` 结尾的串前缀和后缀相等的情况已经找完了，这时需回到最开始，最后再比较一下 `p[0]` 与 `p[x]`，若还不相同，那很显然，以 `p[x]` 结尾的串不存在前缀与后缀相等的情况，`next[x] = 0`

看到这里，估计大家都明白了，**这不就是动态规划吗？**（笑

以上就是 `next[]` 数组的实现思想。

**代码如下：**

```
void get_next(int next[]){
    next[0] = 0;//第一个肯定是0
    int x = 1;//我们从p[1]开始递归
    int now = 0;//next[x-1] = now
    while(x < m){
        if(p[x] == p[now]){
            next[x] = now + 1;//若相等，则直接加一
            now ++;//now也加一计算下一个
            x++;//计算下一个
        }else if(now != 0){//不相等的情况，递归计算次一级的长度
            now = next[now - 1];
        }else{//now = 0 表示上一次循环计算次一级长度的时候不存在，表示找以p[x-1]结尾的串的前缀与后缀相等的情况已经找完了
            //找完了都满足不了p[x] == p[now + 1]这时直接x++进入下一个字符，next[x] = 0
```

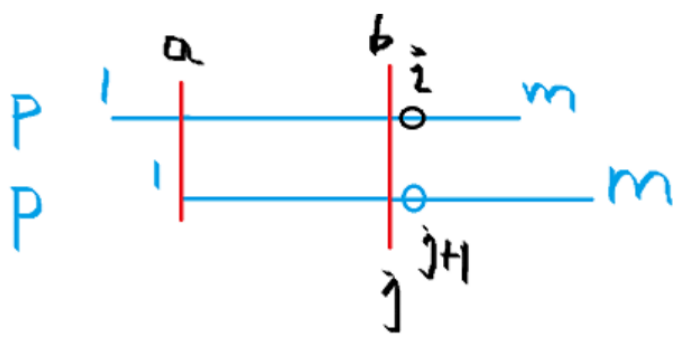
```
                //可以将上面的例子中a串中的字符c改为字符d帮助理解，也可以从x = 1 , now = 0处开始理解
                x++;
            }
        }
}
```

## 第二种`next[]`数组的快速实现：

我们先看代码实现：

```
ne[1] = 0;
for(int i = 2, j = 0; i <= n; i++){
    while(j && p[i] != p[j + 1]) j = ne[j];
    if(p[i] == p[j + 1]) j++;
    ne[i] = j;
}
```

我们先用一个中间过程来理解这段代码，首先要明确，这时规定模式串和目标串的下标都从`1`开始，要不然`j`的初始值不好取：




假设我们已经求得了`next[i-1]`的值，并且`next[i-1]`对应的最长前缀和后缀相等的情况就如上图中的红色区域所示，那么我们再求`next[i]`的时候也会有两种情况：

1. `p[i] = p[j + 1]`
2. `p[i] != p[j + 1]`

对于第一种情况，想法与上文中第一种实现`next[]`时相似，直接`next[i] = next[i - 1] + 1`即可，在这里我们用`j`指针巧妙地表示了`next[i - 1]`的值（从上图中可以看到，`j`的值恰好是`p[i-1]`结尾字符前缀与后缀相等时的最长长度），对应的代码就是：

```
if(p[i] == p[j + 1]) j++;
ne[i] = j;
```

对于第二种情况，当`p[i] != p[j + 1]`时，我们回顾一下上文中第一种实现`next[]`时该怎么做？找以`p[i-1]`结尾的串中前缀与后缀相等的情况下第二长的长度是吧（为什么找第二长的长度上文有详细说明，这里类似），那么以`p[i-1]`结尾的串中前缀与后缀相等的情况下第二长的长度，也就是串`p[a...b]`的后缀与`p[1...j]`的前缀相等情况下的最长长度，也就是串`p[1...j]`的前缀与后缀相等情况下的最长长度，也

就是`next[j]`，分析过程与上文几乎完全一样。对应代码：`while(j && p[i] != p[j + 1]) j = ne[j];`

**几个问题：**

1. 为什么在最开始的时候`i = 2, j = 0`？
    1. `i = 1`的时候很显然，`next[1] = 0`，我们已经求得了`i = 1`的情况下的值，所以`i`要从`2`开始
    2. `j = 0`，其实在这里第二个指针`j`表示的就是上文中的`now`，表示的意思就是：以`p[i-1]`结尾的字符前缀与后缀相等时的最长长度，`i`从`2`开始，`next[i-1]`当然等于`0`
2. 中间过程中`j = 0`的时候表示什么意思？
    1. 对于上面的第二种情况，我们需要找以`p[i-1]`结尾的串中前缀与后缀相等的情况下第二长的长度，若还不满足，继续递归找第三长的长度，一直到`j = 0`，此时意味着以`p[i-1]`结尾的串中前缀与后缀相等的情况已经找完了，我们回退到最开始，用`p[i]`与`p[1](p[j + 1])`进行比较，若相等，则此时刚好就是`p[i]`本身与最开始的字符相等，`next[i] = 1`，否则，不存在`next[i] = 0`
    2. 所以我们看到，在`while`循环中`j`还被赋予次一级长度的值，一直到退出`while`循环，进入下一次`for`循环，`j`重新表示`next[i-1]`

所以我们可以看到第二种`next[]`数组的实现是使用双指针来模拟两个串进行所谓的"匹配"，并保持与KMP算法的一致性，但其实两种实现其实是完全等价的。

我们来对比一下两者的一致性，以下是kmp匹配过程的代码：

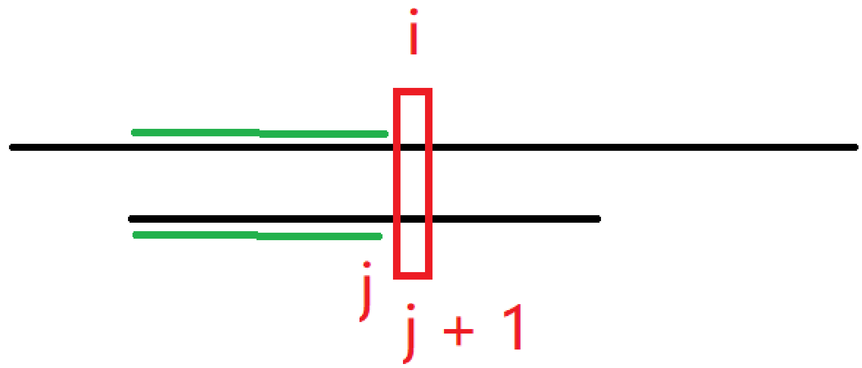
```
for(int i = 1, j = 0; i <= m; i++){
        while(j && s[i] != p[j + 1]) j = ne[j];
        if(s[i] == p[j + 1]){
            j++;
        }
        if(j == n){
            //匹配成功
        }
    }
```

在这里`s[]`表示目标串，`p[]`表示模式串，在匹配过程中仍然有两种可能：

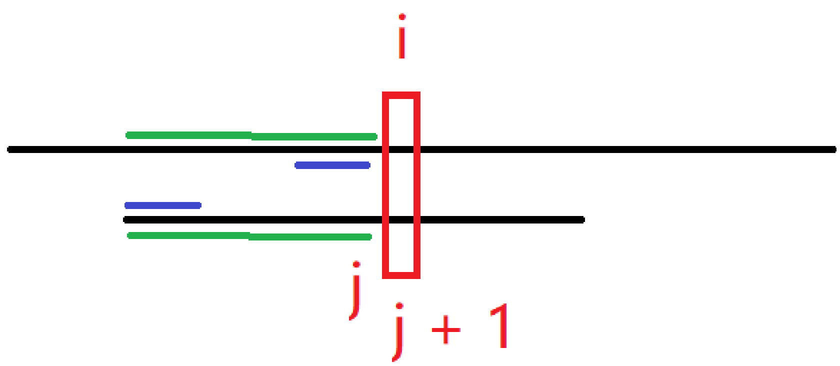
1. `s[i] == p[j+1]`
2. `s[i] != p[j+1]`

当`s[i] == p[j + 1]`的时候，我们直接向后移动`i`和`j`即可，不用再进行其他操作

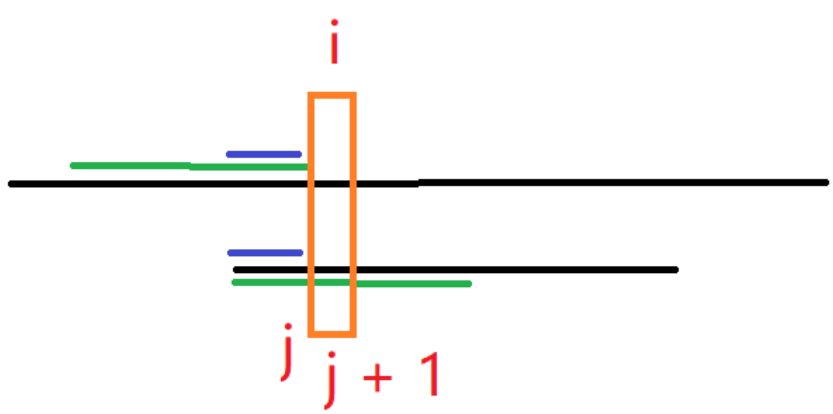
当`s[i] != p[j+1]`的时候，如下图所示，绿色部分代表匹配成功的串，红色代表匹配失败：




在实现`next[]`数组的时候，我们说，此时需要寻找以`p[i - 1]`结尾的前缀与后缀相等情况下第二长的长度，而恰好第二长的长度就是`next[j]`对吧？但是在此处，因为目标串与模式串不一样，也就没有什么第二长的长度这个说法了，我们是直接将`j`回退到`next[j]`，也就是下图中的蓝色部分，跳过注定不可能的尝试（至于为什么不可能，上文有详细说明）：




`j`回退到蓝色部分，然后继续比较`s[i]`和`p[j + 1]`：




`j = 0`的时候表示绿色部分不再有前缀等于后缀的情况，所有的尝试都得跳过，`j`跳到开头重新开始，这时表示`i`开头的尝试直接失败，`i`右移一次，进行下一轮的匹配尝试。

于是我们看到，两种代码虽然很像，但是`j = next[j]`的含义确是不同的，实现`next[]`数组的时候的含义是找到第二长长度的前缀和后缀，匹配的时候是为了跳过注定不可能的尝试并保证不漏尝试（k最大）。

思想都介绍完了（笑，下面是具体的代码实现：

# KMP算法的代码实现：

给定一个字符串 $S$，以及一个模式串 $P$，所有字符串中只包含大小写英文字母以及阿拉伯数字。

模式串 $P$ 在字符串 $S$ 中多次作为子串出现。

求出模式串 $P$ 在字符串 $S$ 中所有出现的位置的起始下标。

**输入格式**

第一行输入整数 $N$，表示字符串 $P$ 的长度。

第二行输入字符串 $P$。

第三行输入整数 $M$，表示字符串 $S$ 的长度。

第四行输入字符串 $S$。

**输出格式**

共一行，输出所有出现位置的起始下标（下标从 0 开始计数），整数之间用空格隔开。

**数据范围**

$1 \le N \le 10^5$
$1 \le M \le 10^6$

**输入样例：**

```
3
aba
5
ababa
```

**输出样例：**

```
0 2
```

---

```
#include<iostream>
using namespace std;

const int N = 100010, M = 1000010;

int n, m;
char p[N], s[M];
int ne[N];

int main(){
    cin >> n >> p + 1 >> m >> s + 1;
    //求next[]数组的过程，从2开始
    for(int i = 2, j = 0; i <= n; i ++){
        while(j && p[i] != p[j + 1]){
```

```
                j = ne[j];//不相等，递归找第二长的前缀和后缀
            }
            //若退出while循环时j = 0，表示所有长度都已找完
            //相等的情况，此时前缀长度++
            if(p[i] == p[j + 1]) j++;

            ne[i] = j;
        }
        //kmp的匹配过程，从1开始
        for(int i = 1, j = 0; i <= m; i++){
            while(j && s[i] != p[j + 1]) j = ne[j];//不相等，递归跳过一定失败的尝试，回退，重新匹配

            //跳出while循环时若j = 0，则表示回退到了起点，重新匹配
            if(s[i] == p[j + 1]){
                j++;//满足相等的话，继续向后尝试，若不满足，则直接i++,进行新一轮的尝试
            }
            if(j == n){
                //匹配成功
                printf("%d ", i - n);//我们存数组是从1开始，但是题目中输出是从0开始，所以不+1
                //目标串可能包含多个模板串，需要反复匹配
                j = ne[j];//j 不能从0开始匹配，因为前面匹配成功的串中记录了信息，需要回退到ne[j]
            }

        }
        return 0;
}
```

# 时间复杂度

我们只分析一部分代码即可, 上面的一部分与这部分完全相同

```
for(int i = 1, j = 0; i <= m; i++){
            while(j && s[i] != p[j + 1]) j = ne[j];//不相等，递归跳过一定失败的尝试，回退，重新匹配
            //跳出while循环时若j = 0，则表示回退到了起点，重新匹配
            if(s[i] == p[j + 1]){
                j++;//满足相等的话，继续向后尝试，若不满足，则直接i++,进行新一轮的尝试
            }
            if(j == n){
                //匹配成功
                printf("%d ", i - n);//我们存数组是从1开始，但是题目中输出是从0开始，所以不+1
                //目标串可能包含多个模板串，需要反复匹配
                j = ne[j];//j 不能从0开始匹配，因为前面匹配成功的串中记录了信息，需要回退到ne[j]
            }

        }
```

首先是`for`循环，最多循环`m`次，对于`j++`这行代码，每次最多加一次，所以在这`m`次循环中，`j`最多加上`m`.下面再看其中的`while`循环。

`while`循环的功能就是把`j`往回跳，而由于最后`j >= 0`所以，在`m`次`for`循环中，`j`最多回跳了`m`次，所以总的复杂度最多是`O(2m)`也就是`O(m)`

从而`kmp`算法的整体复杂度就是`O(n + m)`