KMP详解

有这样一个问题：

给定一个字符串（模式串S）： `"BBC ABCDAB ABCDABCDABDE"`
我想知道其中是否包含（模板串P） `"ABCDABD"` ？

1 两种字符串匹配思路

1.1 暴力思路 O(m * n)

想象两把尺子， `S` 是上面的长尺， `P` 是下面的短尺，最开始两个尺子左端对齐，下面的短尺从左到右一位一位移动，直到匹配成功。
比如：当下面的尺子对齐了上面的 `ABCDAB` ，但是 `D` 对应的是空格，这时匹配失败，只能往右再一位一位移动
暴力实现的代码可以参考如下：

```
char p[N], s[M];
for(int i = 1; i <= m; i ++){
        bool flag = true;
        for(int j = 1; j <= n; j ++){
                if(s[i + j - 1] != p[j]){
                        flag = false;
                        break;
                }
        }
}
```

1.2 KMP思路 O(m + n)

Knuth、Morris、Pratt三个学者提出这样一种方法：就这个例子而言，当 `D` 对应的是空格时，我们并没有前功尽弃，因为我们已知 `D` 前面的 `ABCDAB` 是已经正确匹配的，同时我们发现正确匹配的字符串前后两端都有 `AB` （ `ABCDAB` ），那么我们可以直接向后移动 4 位，继续去匹配空格，观察能否匹配成功，即：

- 初始

> BBC ABCDAB ABCDABCDABDE
> XXX *AB* CD *AB* D

- 向后移动4位

> BBC ABCDAB ABCDABCDABDE
> XXX XXXX *AB* CD *AB* D

- 注：X代表检查过的

如果空格匹配成功则继续，匹配失败的话可以再观察是否有类似“回文”1的形式，如果有的话可以一次移动多位，如果没有只能一位一位移了。（实际上还剩下的AB已经不具备跳着移动的条件了，如果仍然无法匹配，只能全部移过去从头开始了）
注1：“类似”的含义是：ABAB不是回文形式，ABABA是回文形式，两者都可以跳着走。
我们再给一个例子来感受KMP：
模式串S： `ababaeaba`
模板串P： `ababacd`

> *ababa* eaba
> *ababa* cd

`'c'` 无法与 `'e'` 匹配，观察到 `"ababa"` 的头是 `"aba"` ，尾也是 `"aba"` ，那么可以直接移2位

> ab *aba* eaba
>
> xx *aba* bacd

'b'仍然无法与'e'匹配，观察到"aba"的头是"a"，尾也是"a"，那么可以直接移2位

> abab *a* eaba
>
> xxxx *a* babacd

`'b'` 无法与 `'e'` 匹配（紧跟亮色后面的），只剩a也不能跳着走了，这样就只能从e开始重新来了。
如果是暴力的话，第一次匹配不到e，可能就是这样的结果：（一位一位移动看是否能匹配）

- ababaeaba
- xabababacd
- xxabababacd
- …

2 概念补充

2.1 模式串和模板串

s[ ]是模式串，即比较长的字符串。
p[ ]是模板串，即比较短的字符串。（这样可能不严谨）

2.2 前缀后缀

"非平凡前缀"：指除了最后一个字符以外，一个字符串的全部头部组合。
"非平凡后缀"：指除了第一个字符以外，一个字符串的全部尾部组合。
以下均简称为前/后缀。

举例： `P = abcab` ，假设下标从1开始，分别对应1、2、3、4、5

下标	前缀	后缀
1	空	空
2	{ a }	{ b }
3	{ a, ab }	{ c, bc }
4	{ a, ab, abc }	{ a, ca, bca }
5	{ a, ab, abc, abca }	{ b, ab, cab, bcab }

2.3 next数组

在上面的例子中，我们每一次应该向右直接移动多少位是通过next数组得到的，而 `next` 数组是经过预处理得到的。
简单的来讲， `next` 数组可以这样通俗地描述：
`next[i]` ：以i为终点的后缀和从1开始的前缀相等，且长度最长。数组中存放的是这个前缀的最后一个元素的下标。（因为这样才能拼接过去）

```
// 伪代码形式
next[i] = j;
p[1, j] = p[i - j + 1, i]
```

还是刚刚那个： `P = abcab`

下标 i	前缀	后缀	next[i]
1	空	空	0
2	{ a }	{ b }	0
3	{ a, ab }	{ c, bc }	0
4	{ a, ab, abc }	{ a, ca, bca }	1
5	{ a, ab, abc, abca }	{ b, ab, cab, bcab }	2

eg.对5来说， `next[5] = 2` 的含义就是： `p[1 ~ 2] = p[4 ~ 5]`

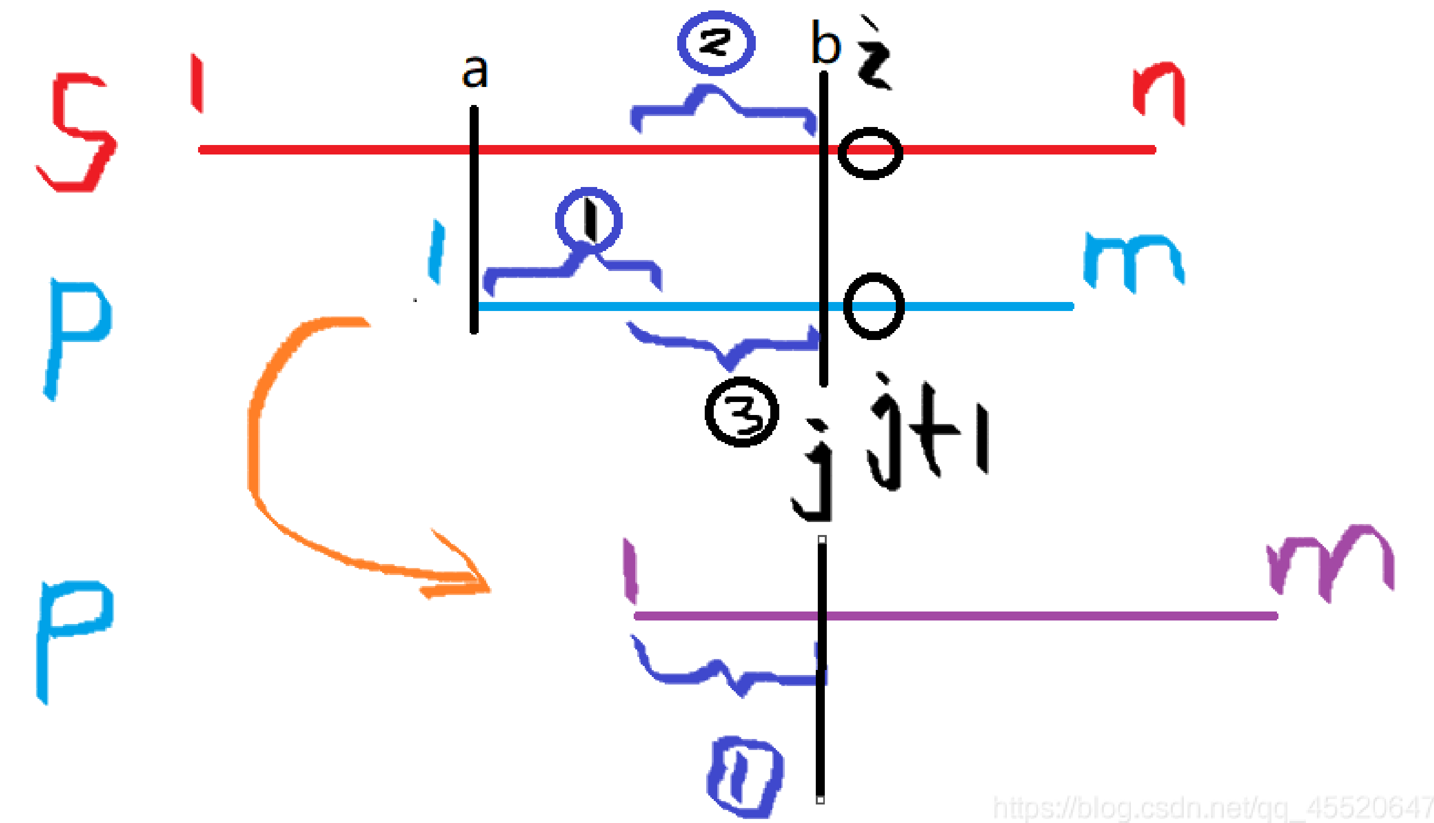
3 代码实现

3.1 next数组的应用

有了next数组，我们每次匹配不成功时，就可以直接得出下一个跳向哪个位置，这一操作可以直接由 `j = next[j]` 来完成。
插句题外话，在 y 总眼里，这是 `j` 的回退，其实 `j` 每次回退，都是 `P` 串在跳跃行进匹配的过程。
假设我们已经有了 `next` 数组，那么 `kmp` 匹配的过程可以用代码表示如下：

```
int ne[N];      // next数组
// kmp 匹配过程
for(int i = 1, j = 0; i <= m; i ++){
        while(j && s[i] != p[j+1]) j = ne[j];   // 当j没有回退起点 且 当前的s[i]无法匹配时
        if(s[i] == p[j + 1]) j ++;
        if(j == n){
                /*
                匹配成功
                */
                j = ne[j];      // 继续寻找下一个匹配串
        }
}
```

贴一张y总上课讲的抽象图：（配合上面代码）



3.2 next数组的初始化

先给一个例子，便于下面算法的模拟，顺便看看对next数组是否理解了：

`P = ababa`，则：（始终记住KMP算法中下标我们都从1开始）

next[i]	初始化	依据
1	0	空
2	0	空
3	1	a
4	2	ab
5	3	aba

next数组的初始化的本质就是自己和自己匹配。
一定要想清楚在kmp匹配过程中，P字符串的 j 到底意味着什么：j 的作用就是下一次 P 字符串的开始匹配点（的前一个）。
通俗的讲，j 就是在P串中能匹配到哪，记录每一个能匹配到哪的位置就是next的初始化过程。
好难描述我脑海中的动图，希望以后的我能看懂……

- 最开始
  j = 0 ababa
  i = 2 baba 无法匹配到，所以ne[2] = 0
- i = 3 aba 匹配到了，j = 1，所以ne[3] = 1
- i = 4 ba 匹配到了，j = 2，所以ne[4] = 2
- i = 5 a 匹配到了， j = 3，所以ne[5] = 3
- 如果匹配不到的话，也不用一步一步挪，直接用已经有的ne[j]，匹配的会更快

代码写出来和3.1几乎一样，毕竟都是匹配的过程。

```
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;
}
```

3.3 总代码

总结：KMP算法可以实现字符串的快速匹配问题
题目链接：KMP匹配字符串

```cpp
#include<iostream>

using namespace std;

const int N = 100010;
const int M = 1000010;
char p[N], s[M];
int ne[N];
int main(){
    int m, n;
    cin >> n >> p + 1 >> m >> s + 1;    // 保证下标从1开始（优雅来自于细节啊）

    // 求next
    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;
    }

    // kmp匹配
    for(int i = 1, j = 0; i <= m; i ++){
        while(j && s[i] != p[j+1]) j = ne[j];   // 当j没有回退起点 且 当前的s[i]无法匹配时
        if(s[i] == p[j + 1]) j ++;
        if(j == n){
                printf("%d ", i - n);
                j = ne[j];      // 继续寻找下一个匹配串
        }
    }
    return 0;
}
```