

KMP

KMP算法（全称Knuth-Morris-Pratt字符串查找算法，由三位发明者的姓氏命名）是可以在**文本串** s 中快速查找**模式串** p 的一种算法。

要想知道KMP算法是如何减少字符串查找的时间复杂度的，我们不如来看暴力匹配方法是如何浪费时间的。所谓暴力匹配，就是逐字符逐字符地进行匹配（比较 $s[i]$ 和 $p[j]$ ），如果当前字符匹配成功（ $s[i] == p[j]$ ），就匹配下一个字符（ $++i, ++j$ ），如果失配， i 回溯， j 置为0（ $i = i - j + 1, j = 0$ ）。代码如下：

```
// 暴力匹配
int i = 0, j = 0;
while (i < s.length()) {
    if (s[i] == p[j]) ++i, ++j;
    else i = i - j + 1, j = 0;
    if (j == p.length()) { // 匹配成功
        // do something
    }
}
```

举例来说，假如 $s = \text{abababcbabaa}$ ，我们暴力匹配，过程会是怎样？

从头开始匹配，第一个字符是a，匹配成功。

↓

abababcbabaa

↑

abababcbabaa

第2~4个字符也匹配成功，继续。

↓

abababcbabaa

↑

abababcbabaa

下一位，匹配失败，回溯。

↓
abab**a**bcabaa
abab**c**abaa
↑

匹配失败，继续尝试。

↓
a**b**ababcabaa
a**a**babcabaa
↑

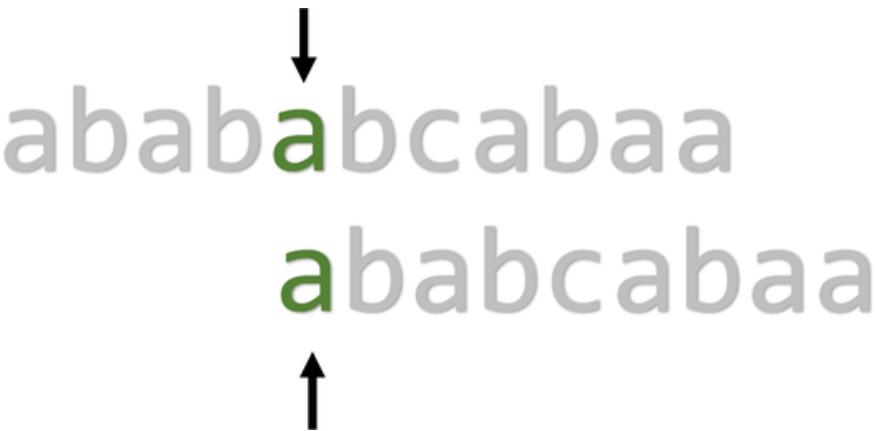
下一位，匹配成功。

↓
ab**a**babcabaa
ab**a**babcabaa
↑

就这样一直匹配到结尾。

↓
abababcabaa**a**
ababcabaa**a**
↑

设两个字符串的长度分别为 n 和 m ，则暴力匹配的最坏时间复杂度是 $O(nm)$ 。究其原因，在于 i 进行回溯浪费了时间。能不能让 i 不走回头路呢？然而，如果 i 不回溯，同时又把 j 置为 0，很可能会出现缺漏，如下图。



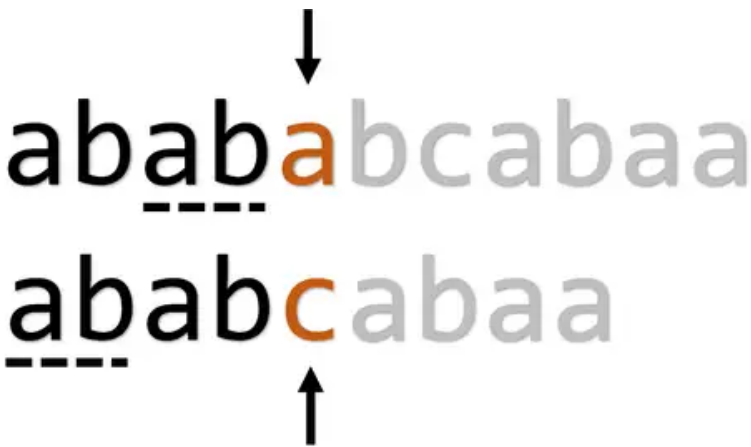
因此，我们考虑让 j 被赋为一个合适的值，我们引入了PMT（Partial Match Table，部分匹配表）。因为 j 的值理论上可以从模式串自身得到，因此我们可以对模式串预处理出来一张PMT表，例如 ababcabaa 对应的PMT如下：

	0	1	2	3	4	5	6	7	8
p	a	b	a	b	c	a	b	a	a
pmt	0	0	1	2	0	1	2	3	1

简单地说， $pmt[i]$ 就是，从 $p[0]$ 往后数、同时从 $p[i]$ 往前数相同的位数，在保证前后缀相同的情况下，最多能数多少位（这也称为前缀的 border）。即：

$$\max\{len\} \text{ where } p[0 \dots len - 1] = p[i - len + 1 \dots i]$$

为什么PMT可以用来确定 j 指针的位置呢？让我们先回到暴力匹配算法第一次失配时的情形：



这时， s 中的 'a' 与 p 中的 'c' 没有配上，我们计划保持 i 指针（上面的指针）不变，而把 j 指针左移。我们注意到，"abab" 已经匹配成功了，它拥有一个前缀 "ab"，以及一个后缀 "ab"（虚线部分），所以我们可以把这个 "ab" 利用起来，变成下面这样：

abababcabaa
ababcabaa

回到我们刚刚的PMT表中，我们发现这时我们正是在令 $j = \text{pmt}[j - 1]$

再举一个例子：

ABACABA BACABAD
ABACABAD

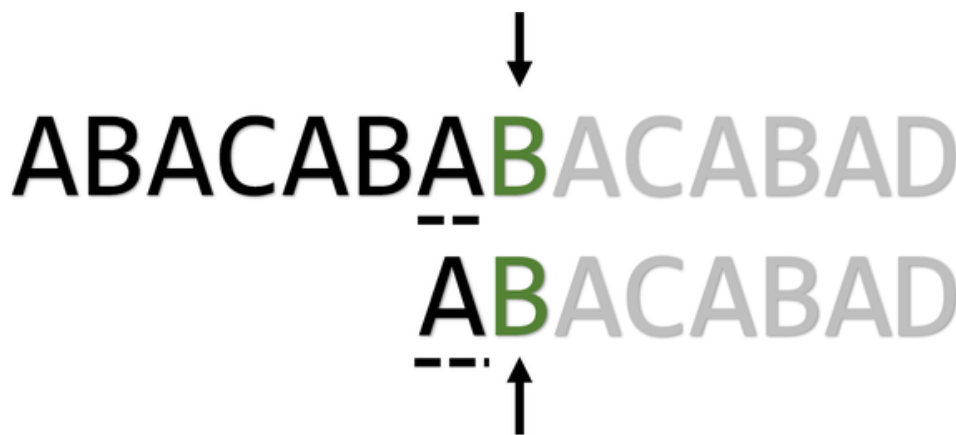
此时发生失配，我们令 $j = \text{pmt}[j - 1]$ ($= 3$) (也就是符合条件的最长前缀所紧接着的下一位)：

ABACABA BACABAD
ABACABAD

仍不匹配，我们继续：

ABACABA BACABAD
ABACABAD

这次取得了成功。当然，我们并不总是能成功，有可能 j 指针一路减到了0，但 $s[i]$ 仍然不等于 $p[j]$ ，这时我们不再移动 j 指针



同时，`pmt` 数组整体右移一位（首位填 `-1`），许多操作会方便许多，我们也通常将其称为 `nxt` 数组

上述的kmp算法转换成代码是这样的：

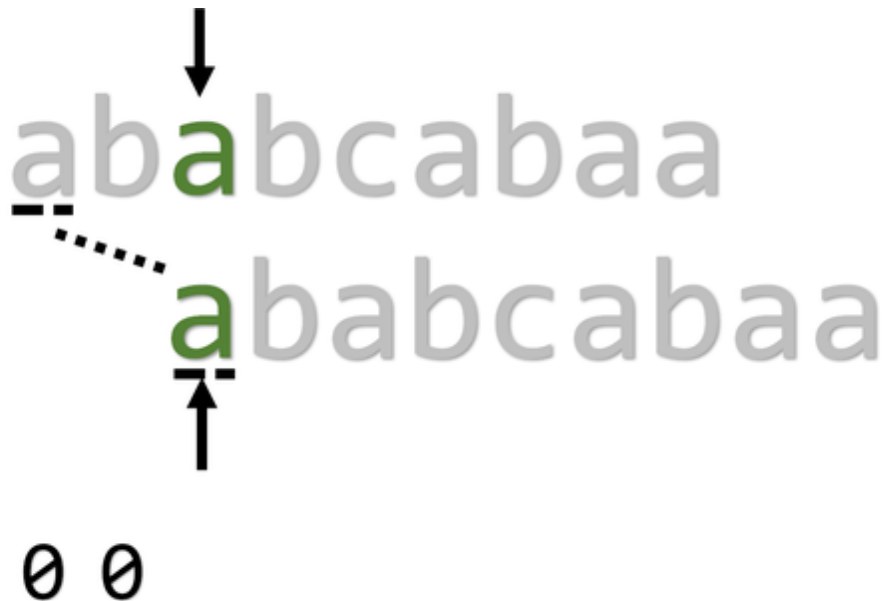
```
void kmp(string& f, string& s, const vector<int>& nxt) {
    int i = 0, j = 0;
    while (i < (int)f.size()) {
        if (j == -1 || f[i] == s[j]) {
            i++; j++;
            if (j == s.size()) {
                // do sth.
            }
        }
        else {
            j = nxt[j];
        }
    }
}
```

现在问题来了，`nxt` 怎么求？如果暴力求的话，时间复杂度是 $O(m^2)$ ，并不理想。一种精妙的做法是，在**错开一位**后，让 `p` **自己匹配自己**（这相当于是用**前缀**去匹配**后缀**）。我们知道 `nxt[0] = -1`，而之后的每一位则可以通过在匹配过程中记录 `j` 值得到。

还是以刚刚的模式串为例：



匹配失败，则 $\text{next}[2] = -1 + 1 = 0$ ， i 指针后移。



接下来匹配成功， j 指针右移，可知 $\text{next}[3] = 1$ ，然后将两个指针都右移。

继续匹配成功， j 指针右移， $\text{next}[4] = 2$ 。

abab**c**abaa
ab**a**bcabaa
0 0 1 2

下一位失配，因为前面的 `next` 已经算出来了，我们可以像匹配文本串时那样地使用它。`next[2]` 即 `pmt[1]` 等于0，所以退回到开头。

abab**c**abaa
ababcabaa
0 0 1 2

`j` 指针已经到了开头，仍未匹配成功，所以不再移动，`next[5] = j = 0`。

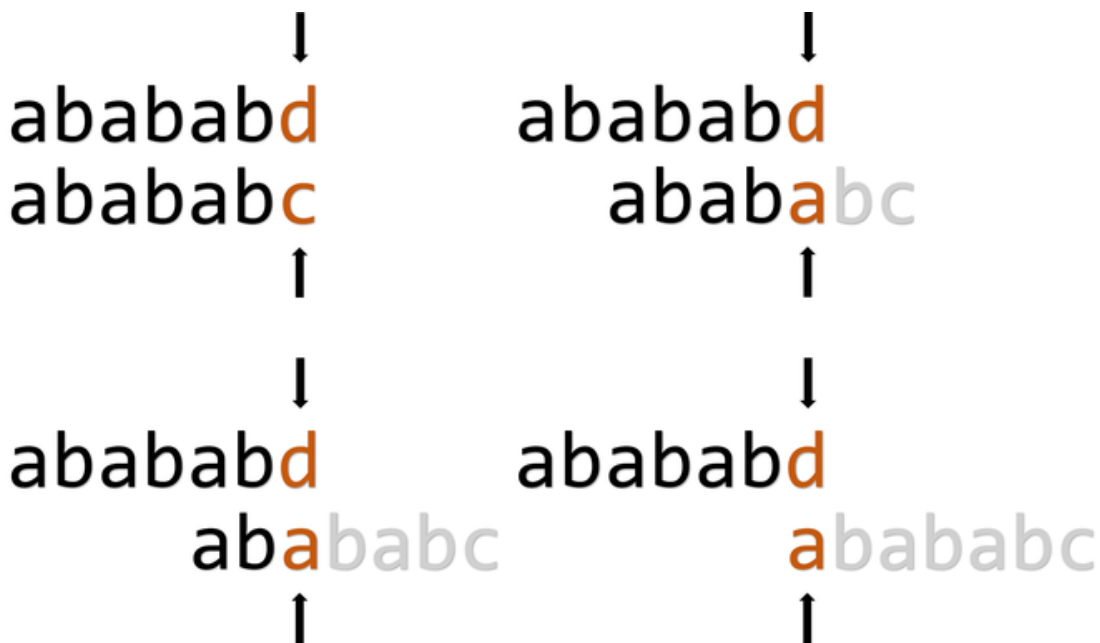
abab**c**abaa
ababcabaa
0 0 1 2

依此类推，转换成代码语言如下：

```
void get_nxt(const string& s, vector<int>& nxt) {
    nxt.resize(s.size());
    nxt[0] = -1;
    int i = 0, j = -1;
    while (i < s.size()) {
        if (j == -1 || s[i] == s[j]) {
            i++; j++; nxt[i] = j;
        }
        else j = nxt[j];
    }
}
```

实际上，上文介绍的算法只能叫MP算法，Knuth将其进行了一个常数优化后，才能称为KMP算法

例如对于 "abababc" 这个模式串，如果我们用它来匹配 "abababd"，在最后处要跳转3次才能发现匹配失败：



其实中间这几次跳转毫无意义，我们明知道 **d** 和 **a** 是不能匹配的，却做了很多无用功。所以我们可以 在计算 `nxt` 时做一些小改动来避免这种情况：

```
void get_nxt(string& s, vector<int>& nxt) {
    nxt.resize(s.size());
    nxt[0] = -1;
    int i = 0, j = -1;
    while (i < s.length()) {
        if (j == -1 || s[i] == s[j]) {
            i++; j++;
            // 相同就一路前跳
            if (s[i] == s[j]) nxt[i] = nxt[j];
            // 不同就不跳
            else nxt[i] = j;
        }
        else j = nxt[j];
    }
}
```



```
}
```

此时的 `next` 数组在部分文章中也称为 `nextval` 数组

复杂度分析

- 求 `next` 数组时候, `i` 指针至多移动 $O(|s|)$ 次, `j` 指针**单次**至多移动 $O(|s|)$ 次, 总共移动 $O(|s|)$ 次
- 在使用kmp算法进行匹配时, `i` 指针至多移动 $O(|p|)$ 次, `j` 指针**单次**至多移动 $O(|s|)$ 次, 总共移动 $O(|s|)$ 次
 - $|s|$ 设为模式串长, $|p|$ 设为文本串长
- 故预处理 $O(|s|)$, 单轮匹配 $O(|s + p|)$

KMP自动机（前缀自动机）

这可能是最简单的自动机算法

我们注意到上述对KMP算法的分析中, **单次匹配**的时间复杂度是 $O(|s|)$ 的 (至多前跳至首部), 于是我们思考, 能否用空间换时间, 在得知下一个字符的情况下, 直接 $O(1)$ 地跳到能匹配到的位置呢?

显然是可以的:

```
// O(|str| * M)初始化, O(|t|)计算str + t中, t部分的pmt函数
// 因为对于pmt[i] <= str.length()的部分, KMP自动机可以直接O(1)转移
namespace KMP_AUTOMATON{
    vector<int> pmt;
    // 字符集大小
    constexpr int M = 26;

    void get_pmt(const string& str){
        pmt.resize(str.length());
        // 获取pmt数组
        for(int i = 1, j = 0; i < str.length(); i++){
            // 不断前跳
            while(j && str[i] != str[j]) j = pmt[j - 1];
            if(str[i] == str[j]) j++;
            pmt[i] = j;
        }
    }

    // res[i][j]表示前i个字符已经匹配上, 在此基础上添加j字符的pmt值
    // 若j == str[i + 1], res[i][j] == i + 1
    void build(string& str, vector<vector<int>>& res){
        // 此处字符串处理根据题意自行分析
        str += '#';
        get_pmt(str);
        int n = str.length();
        res.assign(n, vector<int>(M));
        for(int i = 0; i < n; i++) for(int c = 0; c < M; c++){
            if(i > 0 && 'a' + c != str[i]) res[i][c] = res[pmt[i - 1]][c];
            else res[i][c] = i + ('a' + c == str[i]);
        }
    }
}
```

