# KMP算法

# 1 字符串匹配概念

给定非空字符串$s$和$p$，其长度分别为$n$和$m$，为了便于讨论，将$s$称为主串，$p$称为模式串。查找$p$是否在$s$中存在。

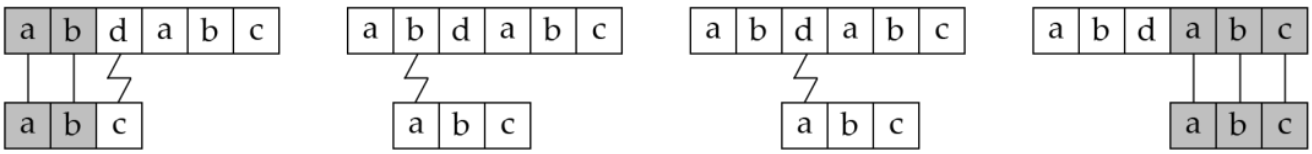
## 1.1 算法

## 1.2 朴素算法

朴素的方法是将$p$的第一个字符与$s$的某个字符对齐，检查对应的字符是否相同， 若从主串的某个位置开始，两者字符全部相同，则发现了匹配。

算法在最坏的情况下时间复杂度为$O(nm)$。

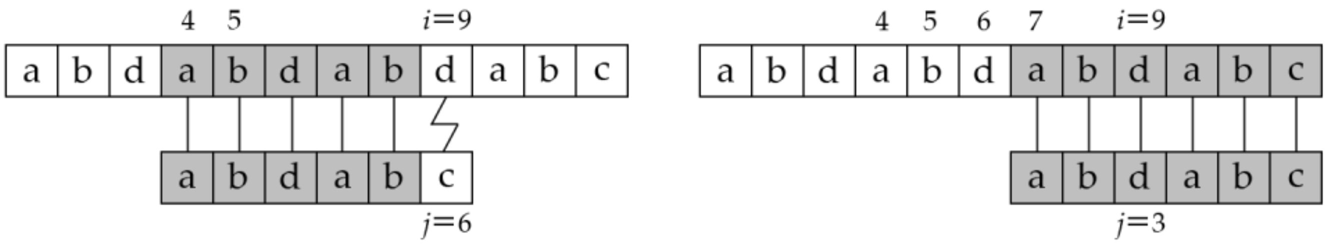
```
bool match(const string &s, const string &p) {
    for (int start = 0; start < s.length(); start++) {
        int i = start, j = 0;
        while (i < s.length() && j < p.length() && s[i] == p[j]) i++, j++;
        if (j >= p.length()) return true;
    }
    return false;
}
```




朴素的字符串匹配过程。$s$ =″abdabc″，p = ″abc″。阴影覆盖的方格为匹配成功的字符，中间使用直线 连接。使用折线连接的方格为最先匹配不成功的字符。观察朴素字符串的匹配过程，不难发现如下规律： 匹配每失败一次，模式串就向右'滑动'一个字符，直到匹配成功或到达主串的末尾。

### 1.2.1 分析原因

朴素的匹配算法之所以在某些情况下效率较低，原因在于每次'失配'后都从模式串的起始处开始重新进行匹配，将之前通过匹配所得到的'额外信息'完全予以丢弃，而这些'额外信息'是能够供后续匹配使用的。 如果善加利用这些'额外信息'，能够有效地提高后续匹配的效率。



失配时'额外信息'的利用。主串$s='abdabdabdabc'$，模式串 $p='abdabc'$，在 $i=9$、$j=6$ 处失配。如果是朴素的匹配算法，下一次应该进行 $i=5$、$j=1$ 的匹配，但是观察模式串 $p$，在失配处 $j=6$ 的字符\c$'$之前有两个字符$'ab'$与模式串的起始两个字符相同，而且已经匹配，那么可以将模式串一次性向右'滑动'3 个字符，跳过 $i=5$、$j=1$，$i=6$、$j=1$，$i=7$、$j=1$ 的匹配，直接开始 $i=9$、$j=3$ 的匹配。也就是说，当 $j=6$ 失配时，可以将模式串位于 $j=3$ 的字符与主串的失配字符对齐，继续进行匹配。因此 $j=3$ 是 $j=6$ 失配时的'跳转'位置，亦即 $j=6$ 失配时，模式串应该向右'滑动'3 个字符，从失配处继续进行匹配。

## 1.3 KMP算法

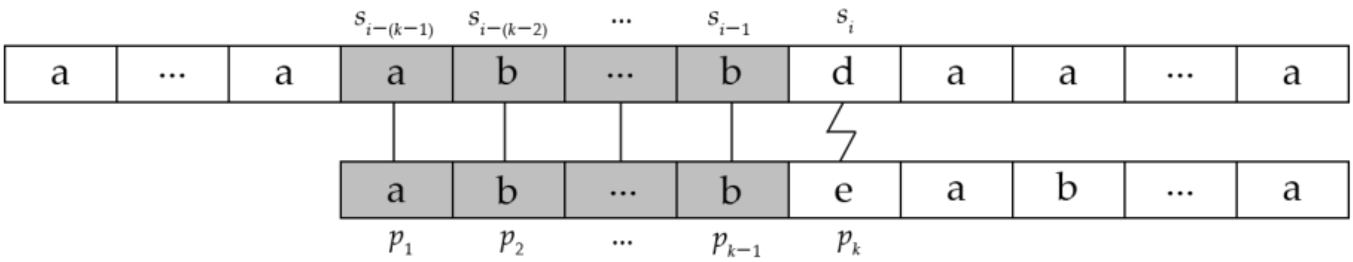
**KMP 匹配算法**由 Knuth、Morris 和 Pratt 各自独立发现，三者合作公布了工作成果。它可以在$\Theta(m)$的时间复杂度内预处理模式串，算法总的匹配时间复杂度为$\Theta(n)$。

假设主串为 $s_1s_2\ldots s_n$，模式串为 $p_1p_2\ldots p_m$，为了提高匹配的效率，需要解决以下问题：

### 1.3.1 失配

当匹配过程中产生'失配'时（即 $s_i \neq p_j$），模式串向右'滑动'的最远距离。也就是说，当主串中第 $i$ 个字符与模式串中第 $j$ 个字符'失配'时，主串中第 $i$ 个字符应与模式串中哪个字符进行再次比较。假设此时应与模式串中第 $k$（$k<j$）个字符继续比较，此时模式串的前 $k$- 1 个字符与主串第 $i$ 个字符之前的 $k-1$ 个字符相同，即模式串中前 $k-1$ 个字符构成的子串必须满足：

$$p_1p_2\cdots p_{k-1} = s_{i-(k-1)}s_{i-(k-2)}\cdots s_{i-1}$$

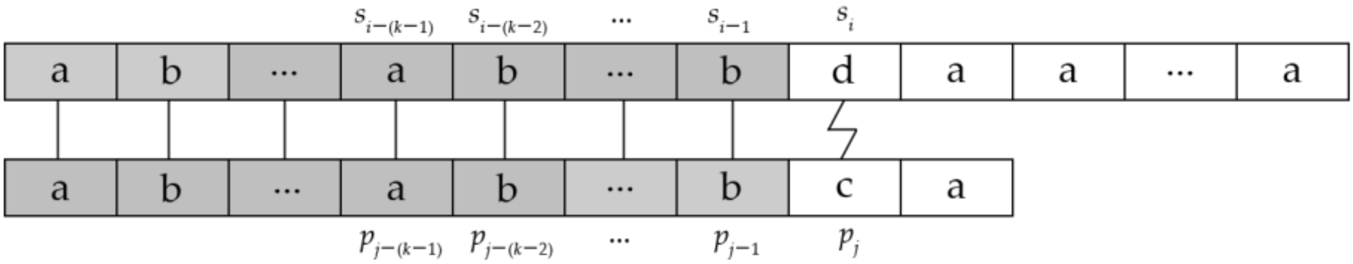



在失配后，将模式串向右'滑动'，假设主串的第 $i$ 个字符与模式串的第 $k$ 个字符开始匹配。易得 $p_1p_2\cdots p_{k-1} = s_{i-(k-1)}s_{i-(k-2)}\cdots s_{i-1}$。

### 1.3.2 部分匹配

在发生失配前，当前主串的部分字符已经跟模式串的部分字符发生匹配，假设发生失配的位置为$s_i \neq p_j$，则有：

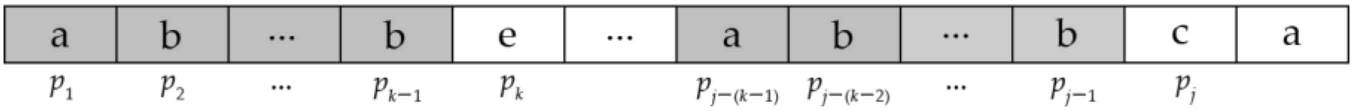
$$p_{j-(k-1)}p_{j-(k-2)}\cdots p_{j-1} = s_{i-(k-1)}s_{i-(k-2)}\cdots s_{i-1}$$



在失配进行'滑动'之前，根据部分匹配的结果，有：
$p_{j-(k-1)}p_{j-(k-2)}\cdots p_{j-1}=s_{i-(k-1)}s_{i-(k-2)}\cdots s_{i-1}$。

由上面两个式子可得：

$$p_1p_2\cdots p_{k-1}=p_{j-(k-1)}p_{j-(k-2)}\cdots p_{j-1}$$




结合失配后向右'滑动'进行匹配的情况及部分匹配的结果，有：
$p_1p_2\cdots p_{k-1}=p_{j-(k-1)}p_{j-(k-2)}\cdots p_{j-1}$。

若令 $next[j]=k$，则 $next[j]$表示当模式串的第 $j$ 个字符与主串中相应字符'失配'时，在模式串中重新和主串中该字符进行比较的字符的位置，由前述讨论可得到模式串 $next$ 函数的定义：

$$next[j]=\begin{cases}0 & j=1\\ \max\{k|1<k<j,p_1p_2\cdots p_{k-1}=p_{j-(k-1)}p_{j-(k-2)}\cdots p_{j-1}\} & \text{当此集合不为空时}\\ 1 & \text{其他情况}\end{cases}$$

由上述定义可以得到模式串$p='abcabcdabcde'$的 next 函数值：

| j | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $p_j$ | a | b | c | a | b | c | d | a | b | c | d | e |
| next[j] | 0 | 1 | 1 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 1 |

在确定模式串 $p$ 的 $next$ 函数之后，匹配可按照以下步骤进行：

1. 假设以指针 $i$ 和 $j$ 分别指示主串 $s$ 和模式串 $p$ 中当前比较的字符，令 $i$ 和 $j$ 的初值均为 $1$ 。
2. 若在匹配过程中，若：

- $s_i=p_j$ ，则 $i$ 和 $j$ 分别自增 $1$；
- $s_i\neq p_j$，$i$ 不变，而 $j$'回退'到 $next[j]$的位置再比较。

3. 重复第2步，直到出现以下两种情形之一：

- j 回退到某个 $next$ 值（$next[next[\ldots next[j]\ldots]]$）时字符比较相等，则指针各自增1继续进行匹配；
- j 回退到值为零（即模式的第一个字符'失配'），则此时需将模式串向右滑动一个位置，即从主串的下一个字符 $s_{i+1}$ 开始，与模式串重新开始匹配。所以此时两个指针都要加1。

## 1.4 next 函数值计算

可以通过递推的方式构造 `next` 数组。

- 把模式串 `p` 拆分成 `l`、`r` 两部分。`l` 表示前缀串开始所在的下标位置，`r` 表示后缀串开始所在的下标位置，起始时 `l = 0`，`r = 1`。
- 比较一下前缀串和后缀串是否相等。通过比较 `p[l]` 和 `p[r]` 来进行判断。

- 如果 `p[l] != p[r]`，说明当前的前后缀不相同。则让后缀开始位置 `k` 不动，前缀串开始位置 `l` 不断回退到 `next[l - 1]` 位置，直到 `p[l] == p[r]` 为止。
- 如果 `p[l] == p[r]`，说明当前的前后缀相同，则可以先让 `l += 1`，此时 `l` 既是前缀下一次进行比较的下标位置，又是当前最长前后缀的长度。
- 记录下标 `r` 之前的模式串 `p` 中，最长相等前后缀的长度为 `l`，即 `next[r] = l`。

举例说明：

主串：ABAABABABCA

子串：ABABC

子串2：ABACABAB

```
void getNext(string &B) {
    int m = B.length();
    Next[0] = 0;
    int i = 1, len = 0;//len为目前最长公共前后缀长度
    while (i < m) {
        if (B[len] == B[i]) {
            Next[i] = ++len;
            i++;
        } else {
            if (len == 0) {
                Next[i] = 0;
                i++;
            } else {
                len = Next[len - 1];
            }
        }
    }
}
```

```
void kmp (string &A, string &B) {
    getNext(B);
    int i = 0, j = 0;
    int n = A.length();
    while (i < n) {
        if (A[i] == B[j]) i++, j++;
        else {
            if (j > 0) {
                j = Next[j - 1];
            } else {
                i++;
            }
        }
        if (j == m)  {
            ans++;
            j = Next[j - 1];
        }
    }
```

```
}
```

## 2 KMP 算法分析

- KMP 算法在构造前缀表阶段的时间复杂度为 $O(m)$，其中 $m$ 是模式串 `p` 的长度。
- KMP 算法在匹配阶段，是根据前缀表不断调整匹配的位置，文本串的下标 `i` 并没有进行回退，可以看出匹配阶段的时间复杂度是 $O(n)$，其中 $n$ 是文本串 `T` 的长度。
- 所以 KMP 整个算法的时间复杂度是 $O(n+m)$，相对于朴素匹配算法的 $O(n*m)$ 的时间复杂度，KMP 算法的效率有了很大的提升。