k 值的确定方法

- Knuth 等人发现，对于不同的 j（P 中的失配位置），k 的取值不同，它仅依赖于模式 P 本身前 j 个字符的构成，与目标无关。
- 可以用一个 next[ ] 失配函数来确定：当模式 P 中第 j 个字符与目标 S 中相应字符失配时，模式 P 中应当由哪个字符（设为第k+1个）与目标中刚失配的字符重新继续进行比较。
- 设模式 $P = p_0p_1\cdots p_{m-2}p_{m-1}$，next [ ] 失配函数定义如下：

$$
\text{next}[j]=\begin{cases}-1, & j=0\\ & 0\le k<j-1 \text{ 且使得}\\ k+1, & p_0p_1\cdots p_k=p_{j-k-1}p_{j-k}\cdots p_{j-1}\\ & \text{的最大整数}\\ 0, & \text{其他情况}\end{cases}
$$

j	0	1	2	3	4	5	6	7
P	a	b	a	a	b	c	a	c
next[j]	-1	0	0	1	1	2	0	1

利用 next 失配函数进行匹配处理

- 若设在进行某一趟匹配比较时在模式 P 的第 j 位失配：
  - 若 $j > 0$，那么在下一趟比较时模式 P 的起始比较位置是 $p_{next[j]}$，目标 S 的检测指针不回溯，仍指向上一趟失配的字符；
  - 若 $j = 0$，则目标 S 检测指针进一，模式 P 检测指针回到 p_0，进行下一趟匹配比较。

运用 KMP 算法的匹配过程如下图

第1趟 目标 a c a b a a b a a b c a c a a b c

模式 a b a a b c a c

× j=1 ⇨ next(1) = 0, 下次p_0

第2趟 目标 a c a b a a b a a b c a c a a b c

模式 a b a a b c a c

× j=0 ⇨ 下次p_0, 目标指针进 1

第3趟 目标 a c a b a a b a a b c a c a a b c

模式 a b a a b c a c

× j=5 ⇨ next(5) = 2, 下次p_2

第4趟 目标 a c a b a a b a a b c a c a a b c

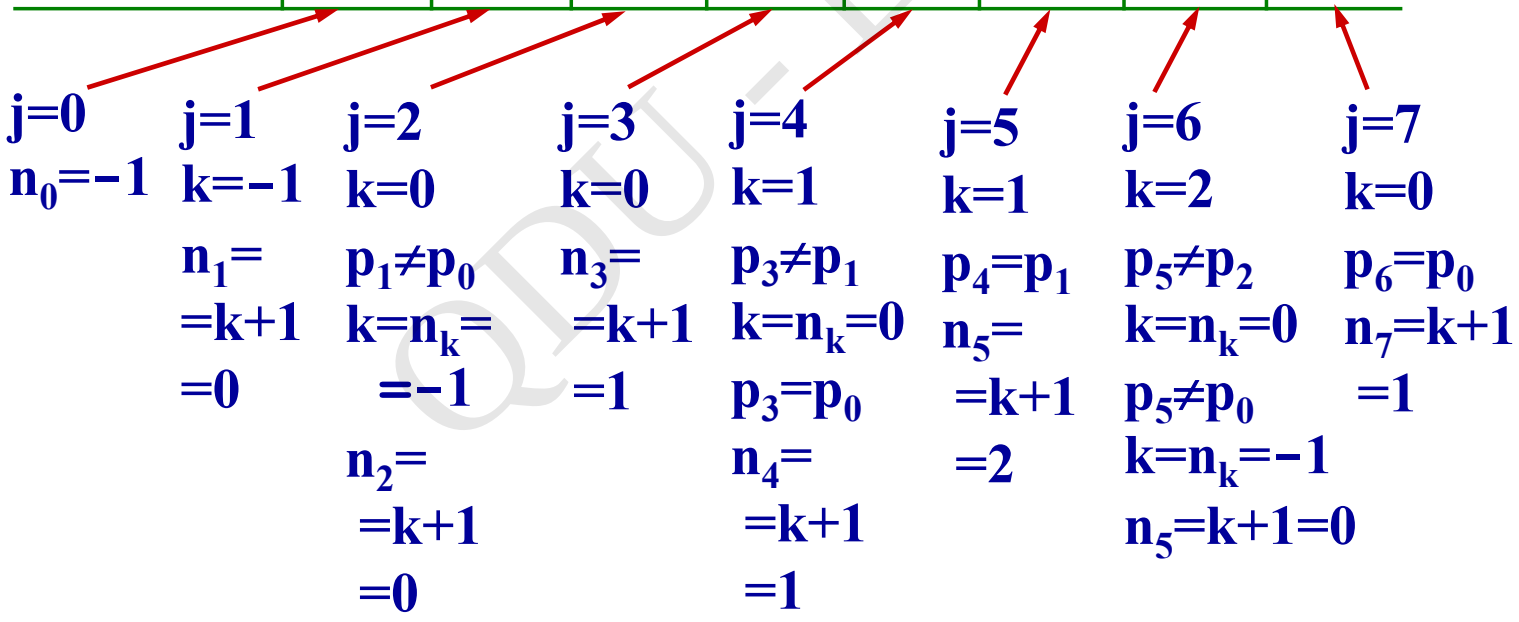
模式 (a b) a a b c a c √

next 失配函数的计算

- 设模式 $P = p_0 p_1 p_2 \cdots p_{m-1}$ 由 m 个字符组成，而next 失配函数为next $= n_0\, n_1\, n_2 \ldots n_{m-1}$，表示了模式的字符分布特征。
- next 失配函数从0, 1, 2, …, $m-1$逐项递推计算:
  - 当 $j = 0$时，$n_0 = -1$。设 $j > 0$ 时 $n_{j-1} = k$:
  - 当 $k = -1$或 $j > 0$且 $p_{j-1} = p_k$，则 $n_j = k+1$。
  - 当 $p_{j-1} \neq p_k$ 且 $k \neq -1$，令 $k = n_k$，并让③循环直到条件不满足。
  - 当 $p_{j-1} \neq p_k$ 且 $k = -1$，则 $n_j = 0$。

- 以前面的例子说明：

j	0	1	2	3	4	5	6	7
P	a	b	a	a	b	c	a	c
next [*j*]	−1	0	0	1	1	2	0	1



由模式串t求next值的算法：

```
void GetNext(SqString P, int next[])
{
    int j, k;
    j = 0;
    k = -1;
    next[0] = -1;
    while(j < P.length - 1) {
        if(k == -1 || P.SString[j] == P.SString[k])
        {
            j++;
            k++;
            next[j] = k;
        }
        else
            k = next[k];
    }
}
```

KMP算法

```
int Index_KMP(SqString S, SqString P, int pos)
{
    int next[MaxSize], i = pos-1, j = 0;
    GetNext(P, next);
    while(i < S.length && j < P.length) {
        if(j == -1 || S.SString[i] == P.SString[j]) {
            i++;
            j++;        //i、j各增1
        }
        else
            j = next[j];    //i不变，j后退
    }
    if(j >= P.length)
        return(i - P.length); //返回匹配模式串的首字符下标
    else
        return -1;          //返回不匹配标志
}
```

算法分析

- 此算法的时间复杂度取决于 **while** 循环。由于是无回溯的算法，执行循环时，目标 **S** 字符比较有进无退，要么执行 **i++** 和 **j++**（对应位相等），要么查找 **next[ ]** 数组进行模式 **P** 位置的右移，然后继续向后比较。字符的比较次数最多为 $O(n)$，n 是目标 **S** 的长度。

END