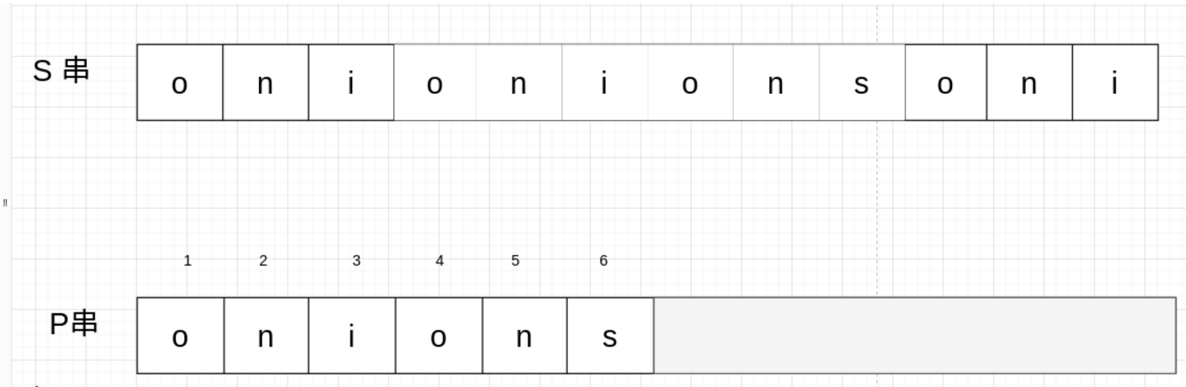
字符串匹配

「 KMP 算法」

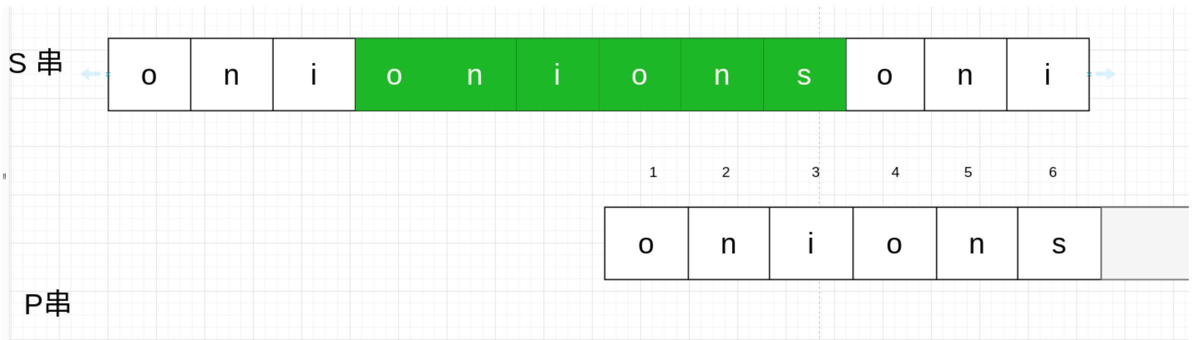
算法分析讲解：

首先我们有 S串（原始串）， P串（模拟串）， 我们在这里默认字符串的下标是从1开始的。如果我们在 S串 挨个字符一个一个去匹配，会花费大量的时间， 时间复杂度 $O(m*n)$，n 为 S 串长度， m 为 p 串长度。

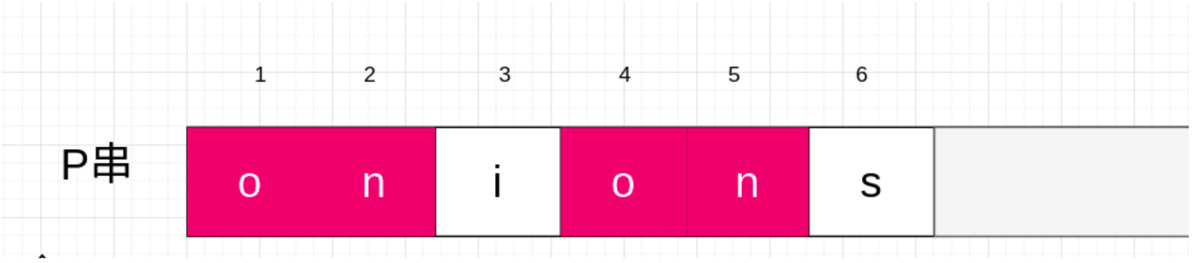


那么节省时间最好的方式就是 `跳过目前看起来肯定用不着` 的字符，比如上图当下标为6的字符 `s` 与原始串的 `i` 不相等，那么下一次 P 串开头匹配的位置绝对不会是在 S 串下标 2， 3（对应着 n，i）处。

那该跳跃多少？是跳跃 P 串的长度 m 吗？ 那就会跳过了正确答案。



那么我们再回来观察下这个 P 串有什么特点。我们发现 P 串的前半部分有两个字符和后面的两个字符相同。这说明之前那种匹配失败的情况下，P 串的开头可以出现在 S 串下标为 4 的地方。



这里引入一个概念： `相等最长前后缀` 来表示这种现象。

代代

求相等最长前后缀：

`kmp` 算法的难点就是在于如何求解相等最长前后缀。

在程序中我们用 `next` 数组来表示 P 串的最长前后缀的关系，这里 `next` 的含义表示的不是下一个而是 `上一个` 如图：

	1	2	3	4	5	6
P串	o	n	i	o	n	s
Next数组	0	0	0	1	2	0

针对 P 串，我们有两个指针 j, i 分别指向前缀末端和遍历的字符，遵循以下的规则：

$$initial : i = 2, j = 1$$

关于 j 我们有：

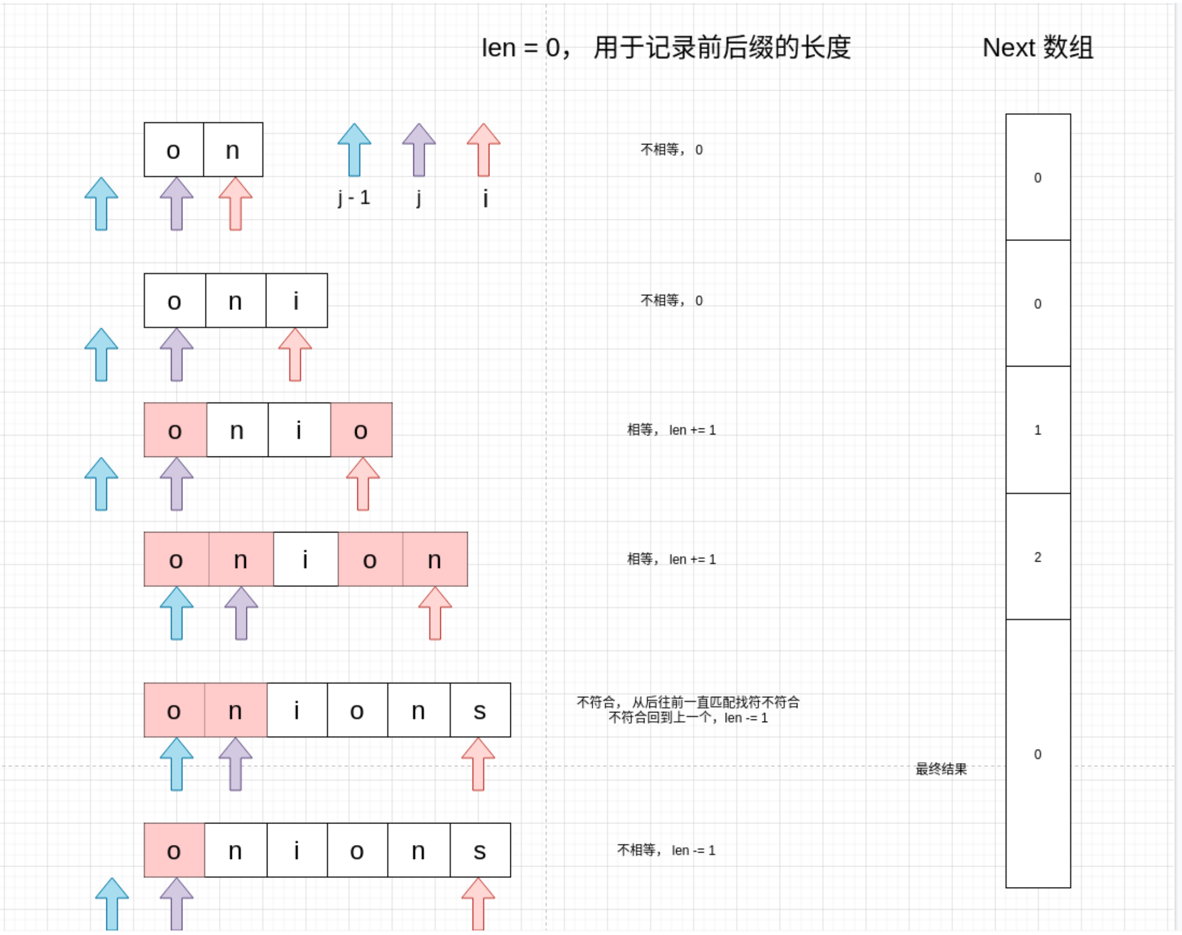
$$j = \begin{cases} j+1, & (P_i = P_j) \\ next_{j-1}+1, & (P_i \neq P_j, j > 1) \end{cases}$$

关于 next 数组我们有：

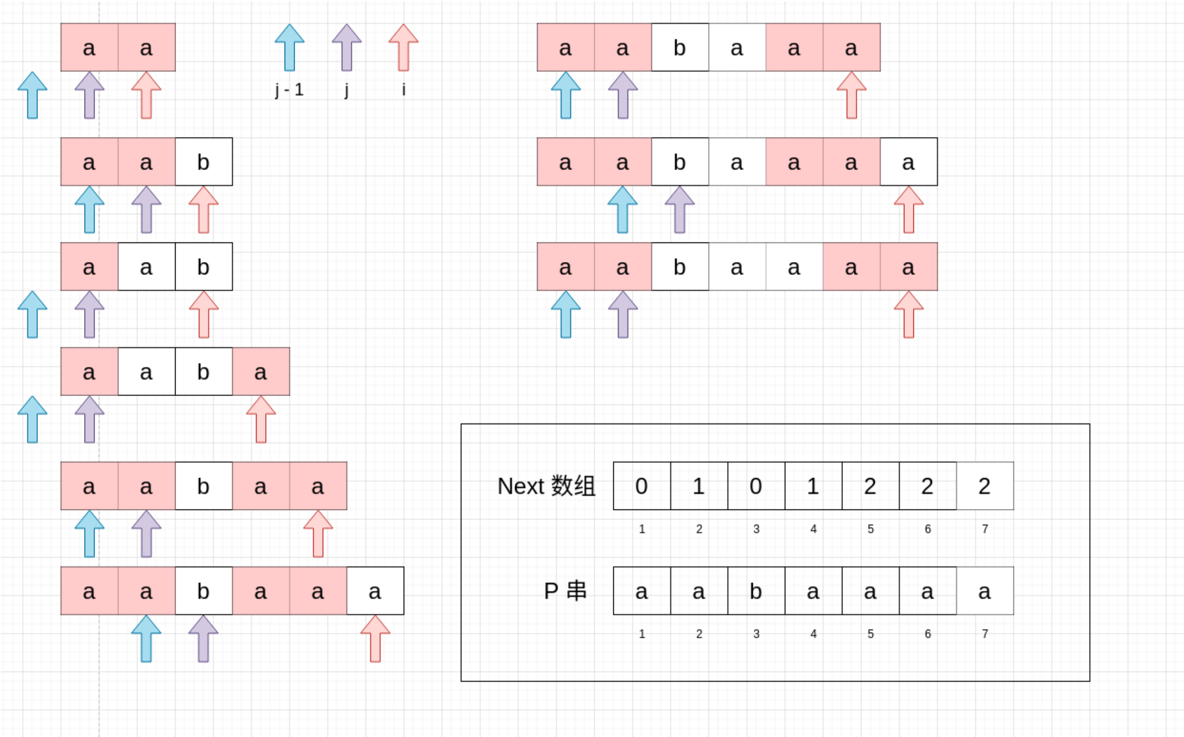
$$next_i = \begin{cases} 0, & (P_i \neq P_j) \\ j, & (P_i = P_j) \end{cases}$$

$j = next_{j-1} + 1$ 原式为： $j - 1 = next_{j-1}$ 这一步表示的是 j 回到前缀末端的前一个元素，或者说是缩短现在的前缀。而 i 就是一个遍历元素， 但是在朴素的 `kmp` 写法里有 `i --` 充当循环的作用。

下图记录了 `next` 数组产生过程：



这里还有些数据样例分析，例如 `aabaaaa` ：



还有很多特殊的数据还没列出来比如 `nenenw` ，其中 `nen` 这类会重叠的字符串也算最长相等前后缀的一种， 因为前后缀只要满足字符串不是只有两端的单个字符即可。

那么我们就可以来写构造 `next` 数组的代码了。

朴素版代码分析：

```
void buildNext(int ne[], int m, char p[]) {
    for (int i = 2, j = 1; i <= m;i ++) {
        if ( p[i] == p[j] )
            ne[i] = j ++;
        else {
            if ( j > 1 ) {
                j = ne[j - 1] + 1;    //把 j 向前移到前一个next记录的位置，终点是开头
                i --;    //i -- 相当于是一个内循环
            }
            ne[i] = 0;   //如果 j 到了开头还没令 p[i] == p[j]，那么他的next就是 0 了
        }
    }
}
```

我们在写朴素版的时候发现如果我们假设从 1 开始的 j 项去和 i 项去比较，不管是公式还是代码会显得有些鸡肋。这时候我们不妨就可以假设从 1 开始的 j + 1 项去和 i 项比较，从而来优化代码，进而得到了y总课上的求next的代码：

nb版代码分析：

```
void buildNbNext(int ne[], int m, char p[]) {
    for (int i = 2, j = 0;i <= m;i ++) {    //设j + 1 从 1 开始
        while ( j && p[i] != p[j] )  j = ne[j];  //如果 j + 1 不在开头并且不相等， j向前移动
        if ( p[i] == p[j] )  j ++;
        ne[i] = j;  //next拿到的是 j + 1 的值或者是 0 值， 由前面的if和while决定
    }
}
```

字符串匹配分析：

```
既然我们已经得到了`最长的前后缀`的关系（next 数组），接下来就该实现之前的设想的方式了。
```

实现的原理方法与求解 `next` 数组类似。

针对 S 串和 P 串， 我们有 i， j两个指针， 为了编写方便也是从 j + 1 项与第 i 项比较， 所以 i 指针负责遍历 S 串的每个字符， j + 1 指针负责指向 P 串目前所匹配到的进度。

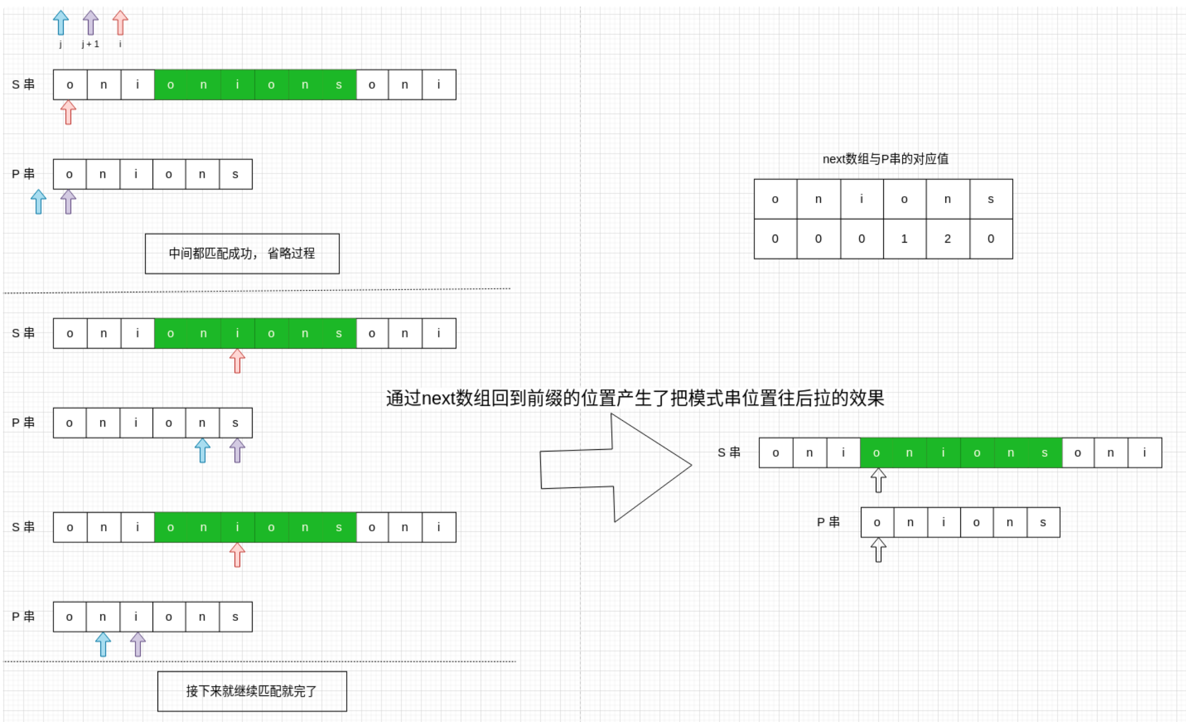
遵循以下规则：

$$initial : i = 1, j = 0$$

对于 j 有：

$$j = \begin{cases} ne[j], & ((s[i] \neq p[j+1], j \neq 0) \quad or \quad j = n) \\ j + 1, & (s[i] = p[j+1]) \end{cases}$$

下图记录了匹配的过程：



匹配代码分析：

```c
void matchString(char p[], int m, char s[], int n, int ne[]) {
    for (int i = 1, j = 0;i <= n;i ++) { //假设j + 1 从 1 开始和 i 项比较
        while ( j && s[i] != p[j + 1] )   j = ne[j]; //j + 1 回到next的前状态
        if ( s[i] == p[j + 1] )  j ++;
        if ( j == n ) {
            printf("%d ", i - m);   //模式串在原始串中的起始位置下标（from 0）
            j = ne[j]; //继续匹配;
        }
    }
}
```

总结：

通过以上的分析我们可以了解到 `kmp` 算法是由 `最长相等前后缀求解` 和 `字符串匹配` 两个部分组成， 后者依赖于前者才能使时间得到减少，总体的时间复杂度是 $O(m+n) = O(n)$

```c
void kmp(char p[], int m, char s[], int n, int ne[]) { //默认下标从 1 开始
    buildNext(ne, m, p);
    matchString(p, m, s, n, ne);
    //好吧这样写纯属就是懒←__←
}
```