数据结构与算法 (Python)

KMP Algorithm

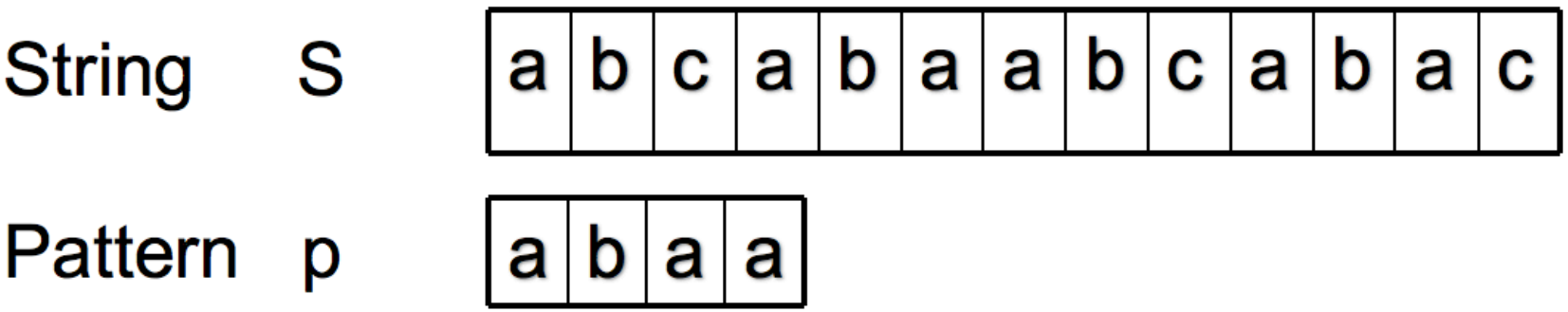
谢正茂 webg@PKU-Mail

北京大学计算机系

April 15, 2021

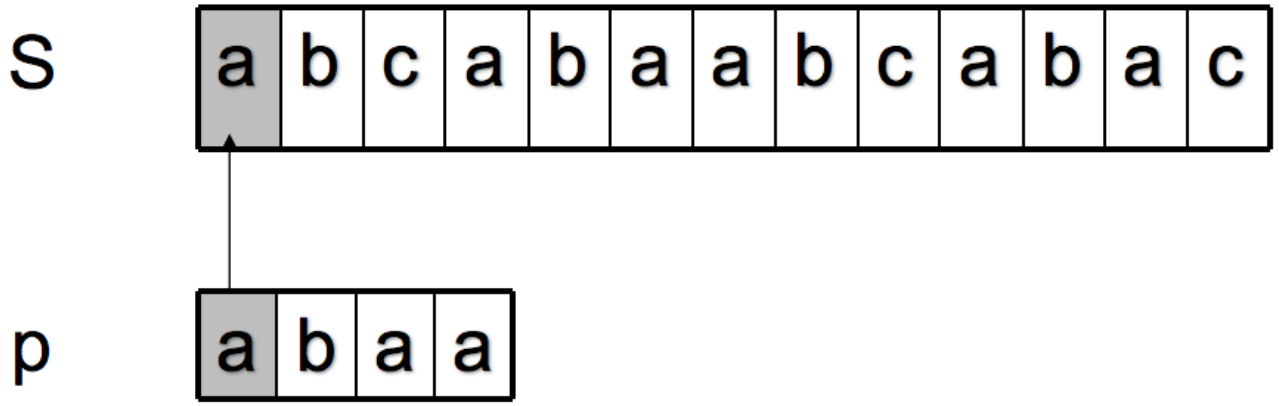
字符串查找问题

- 问题的定义：从一个较长的字符串 S(string，长度 n) 中，查找一个较短的字符串 P(pattern，长度 m) 有没有出现，如果有的话，返回 P 在 S 中出现的位置。
- 计算机上使用最频繁的操作之一
- 首先想到一个 O(mn) 的查找算法
  - 把 S 和 P 左对齐，逐字比较 S 和 P
  - 如果发现有不匹配的地方，把 P 向右移动一格再重新开始
  - 直到发现 P 在 S 中出现了一处完全匹配

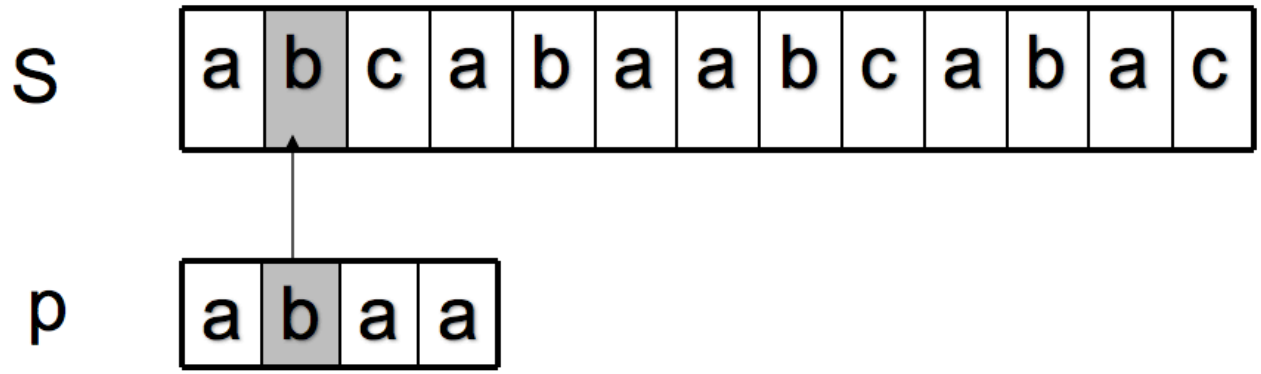


O(mn) 算法

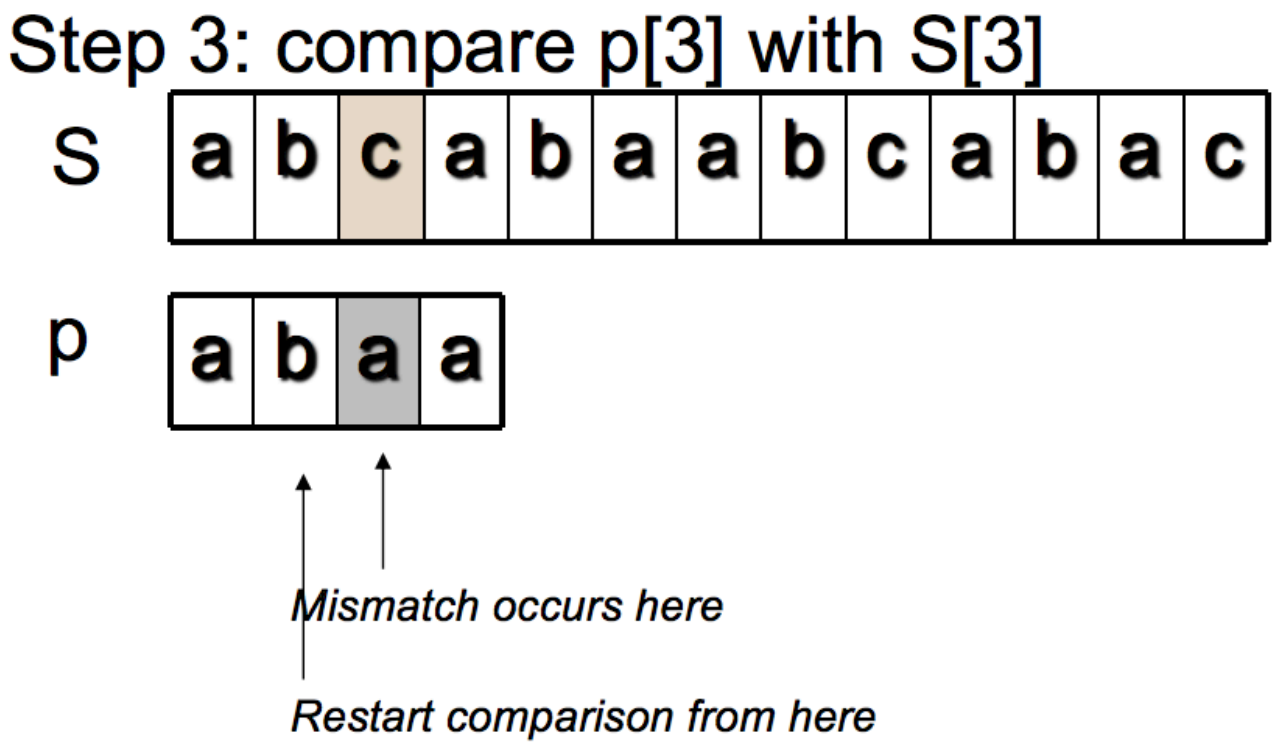
Step 1: compare p[1] with S[1]



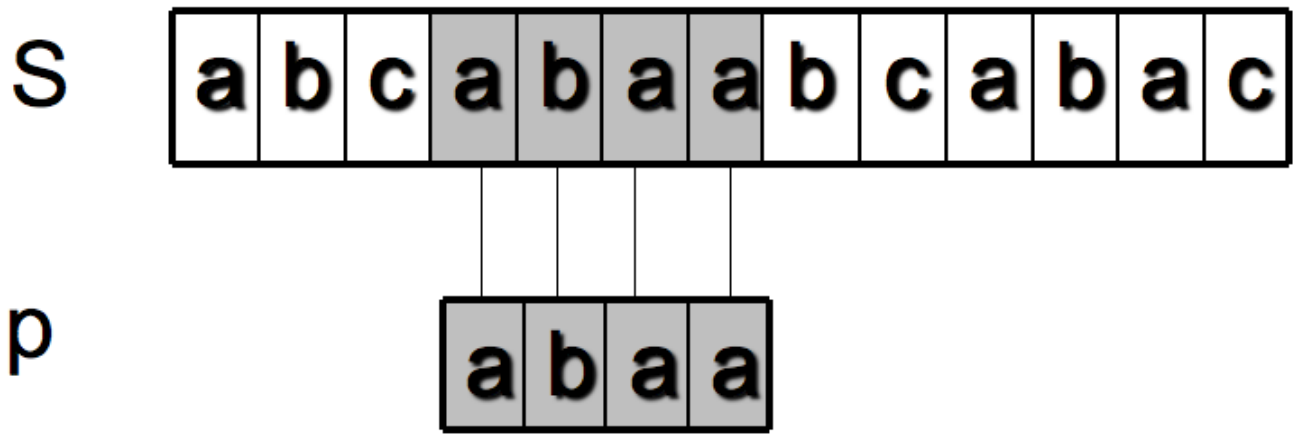
Step 2: compare p[2] with S[2]



O(mn) 算法


Step 3: compare p[3] with S[3]
S
a b c a b a a b c a b a c
p
a b a a
Mismatch occurs here
Restart comparison from here

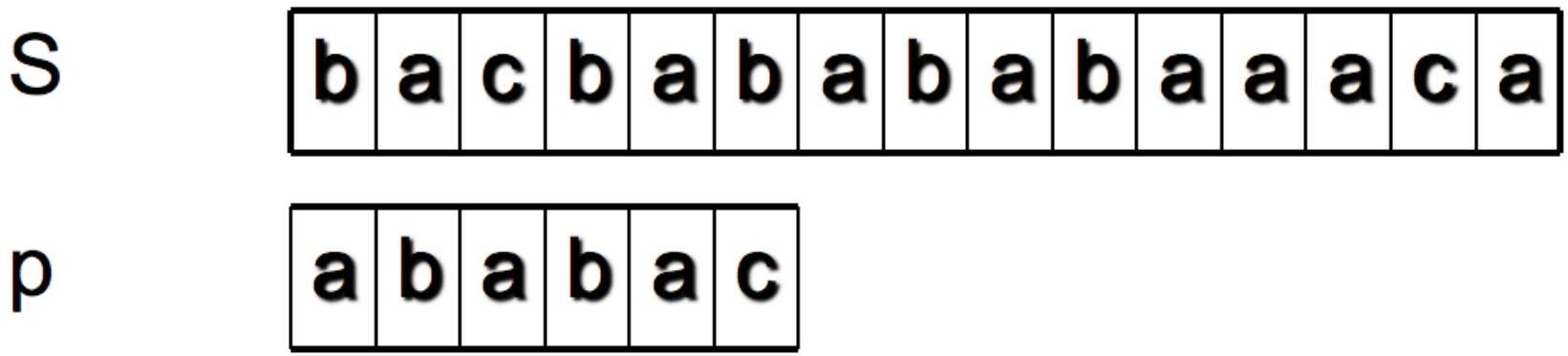
O(mn) 算法



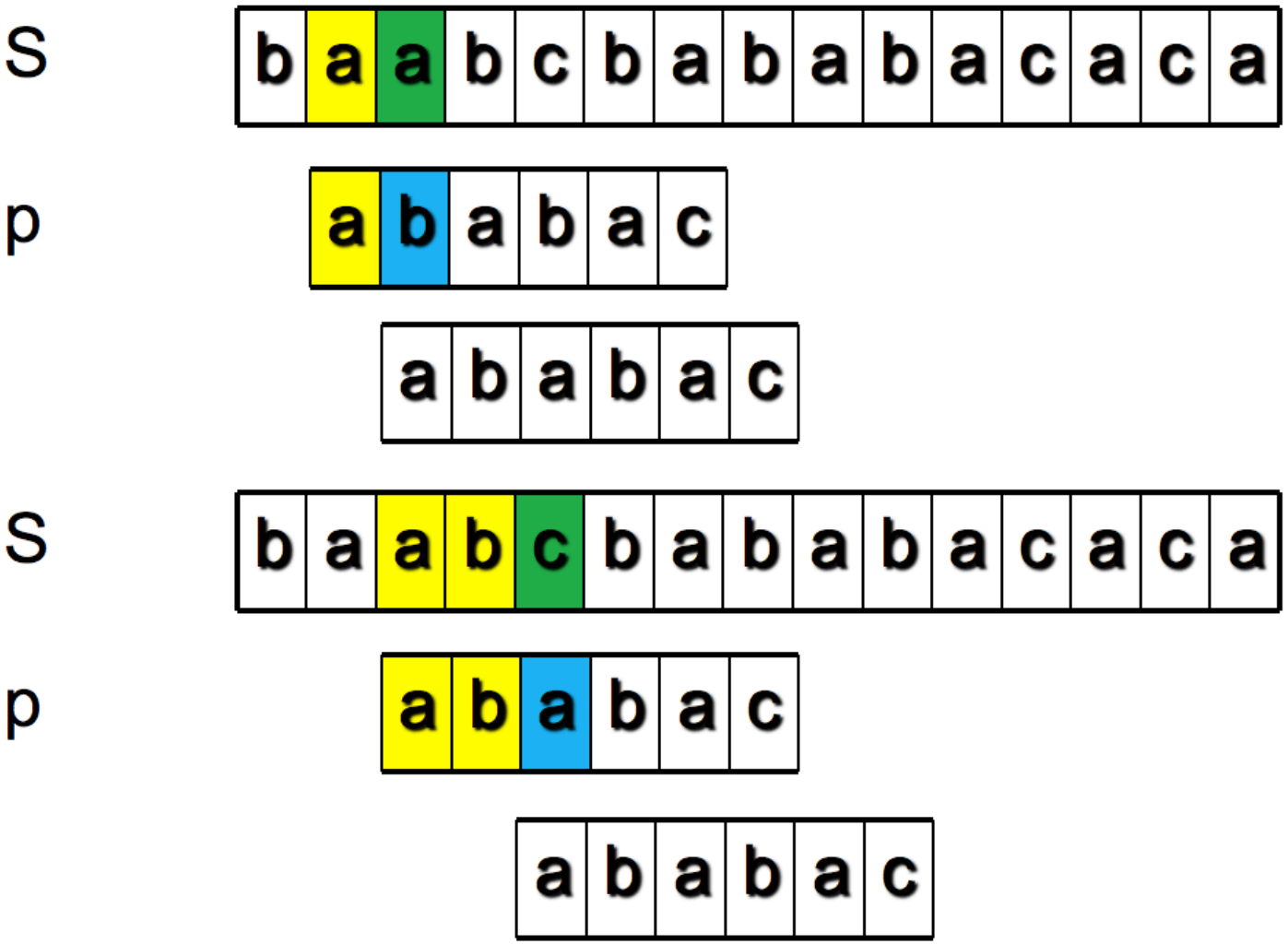
- 在上面的例子中，P 经历了三次右移之后，在 S 中找到了一处它的完全匹配
- 在最坏的情况下，P 可能经历 n-m 次移动，共进行了 n-m+1 轮比较，每轮比较 m 个字符，所以复杂度为 O(mn)

The Knuth-Morris-Pratt Algorithm(KMP)

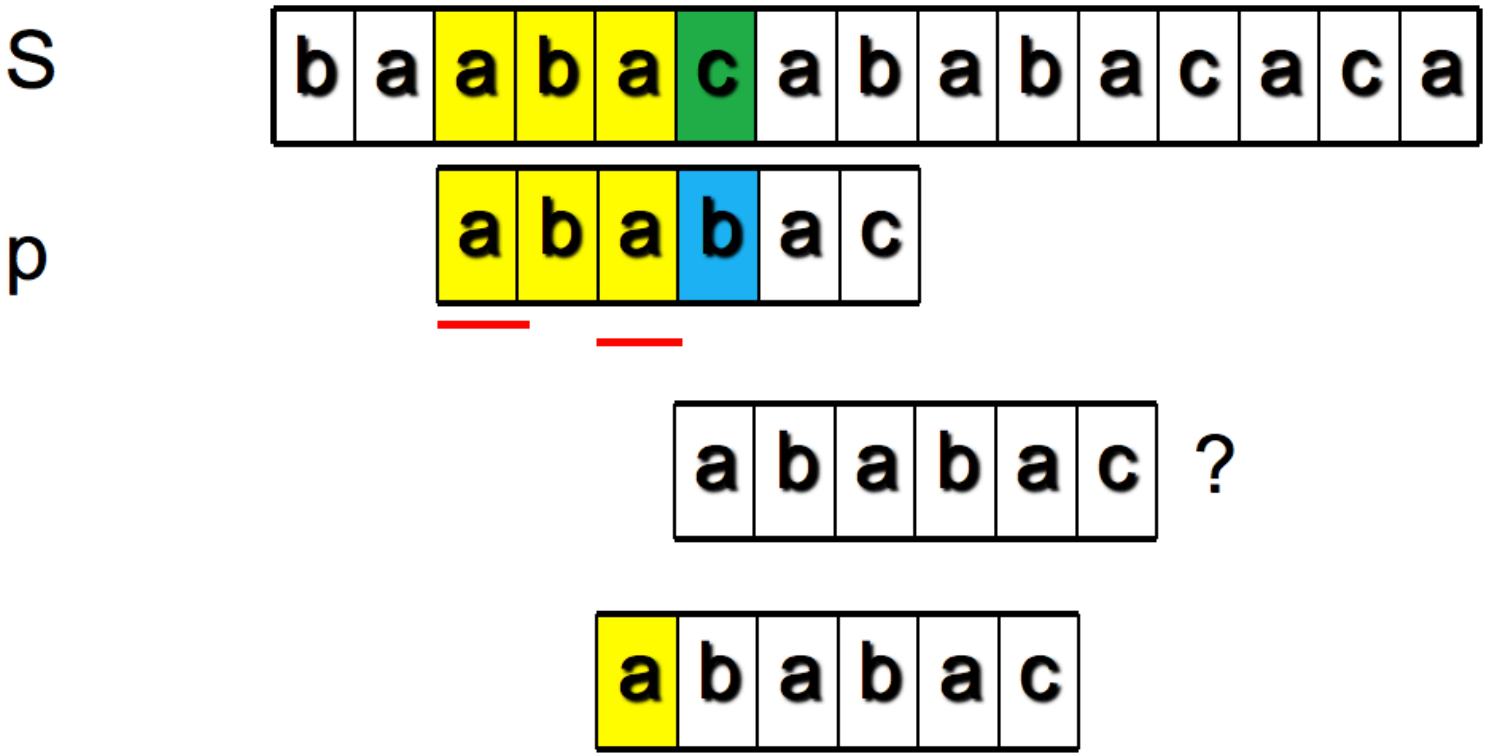
- 改进上面的 O(mn) 算法，Knuth, Morris 和 Pratt 设计了一个线性时间算法。
- 新算法获得了 O(n+m) 的复杂度：当一次不匹配出现 P[i] 与 S[j] 之间时，P[:i] 与 S[j-i:j] 应该是匹配的。通过事先对 P 进行计算，来实现 P 的右移（不再是一格），来保证 S[:j] 中的字符不会再次被比较到。
- 我们通过一些 S 和 P 的例子来进行说明：



观察


S
b a a b c b a b a b a c a c a
p
a b a b a c
a b a b a c
S
b a a b c b a b a b a c a c a
p
a b a b a c
a b a b a c

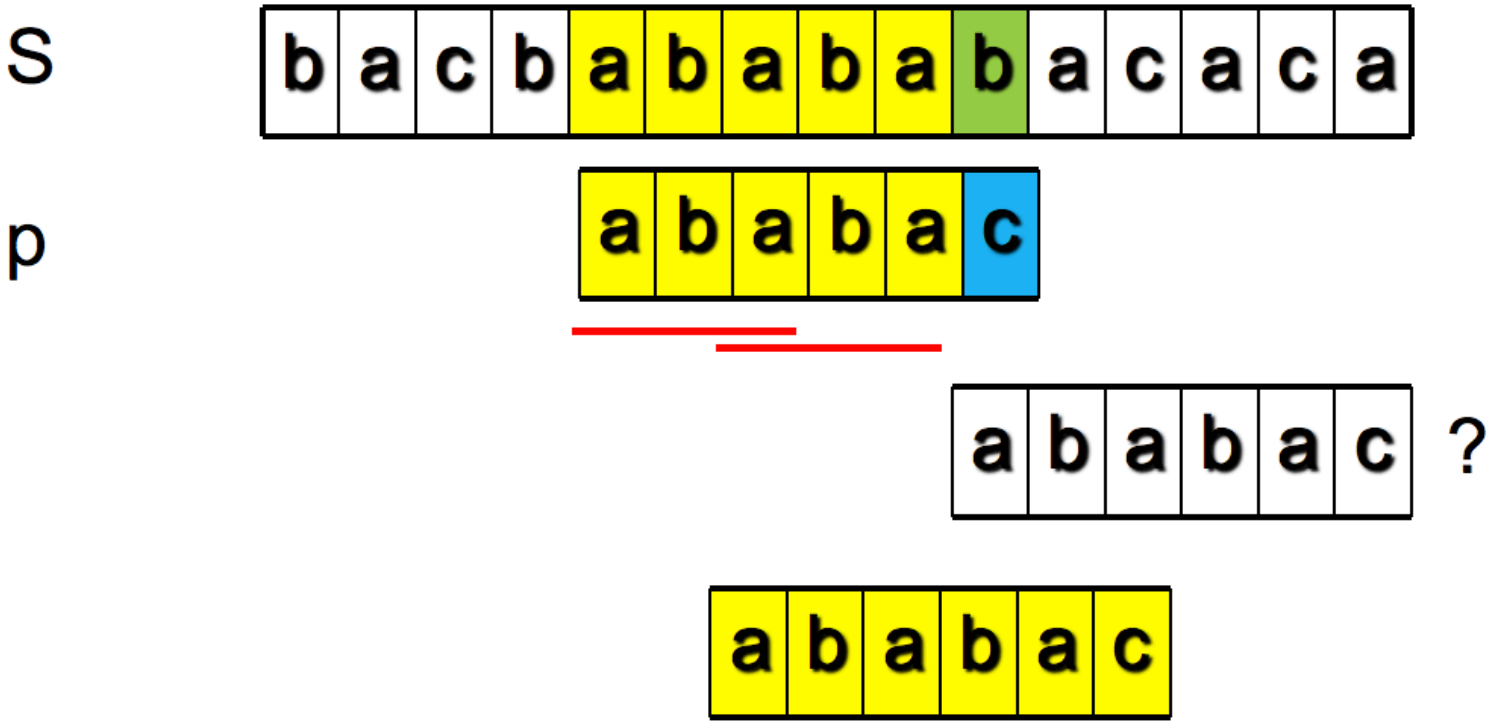
观察



- 关键每次比较失败以后，应该往前挪多少?
- 这取决于 P 长成什么样子

再仔细看

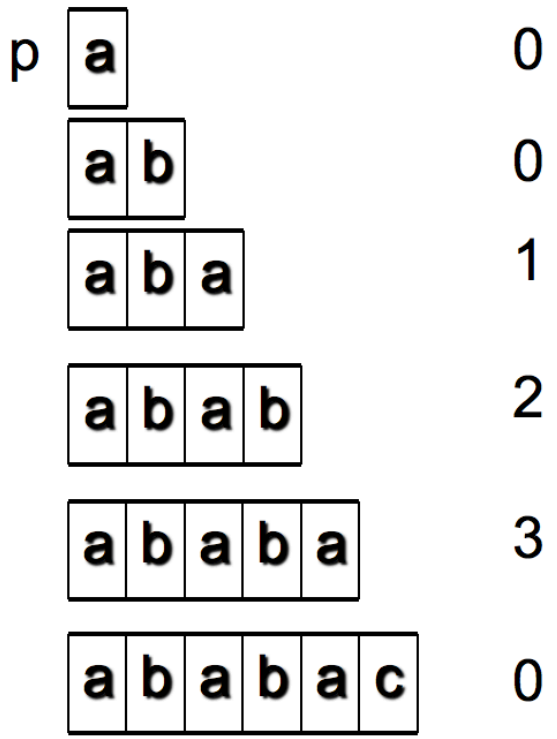
最长公共前后缀为3，指针直接前移3就可以，前面不需要再比较



- P 长成什么样子
- 关键是黄色的部分 P[:i] 有多少前缀的和后缀的是相同的

再睁大眼睛看

- 对于所有的 i，我们需要知道 P[:i] 中从头多少个 (前缀 prefix) 和从尾巴多少个 (后缀 postfix) 是一样的（不能看全部的）



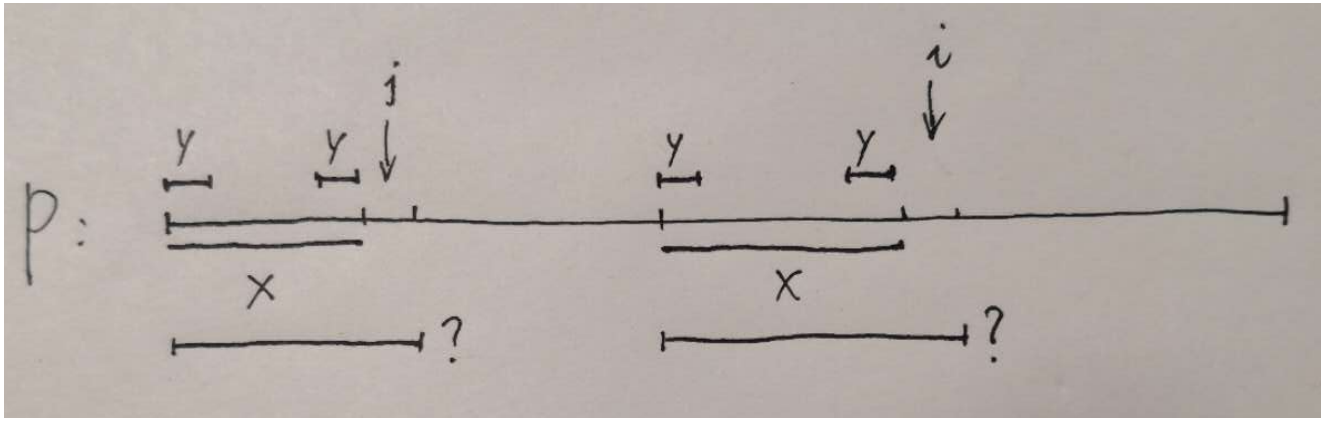
前缀是除了最后一个字符外的所有子串，后缀是除去第一个字符外的所有子串

- 把右边的值取个名字，叫做字符串的最长公共前后缀（长度），表示为 $\max_{ps}(P[:i]), 1 \leq i \leq 6$

定义函数 partial(pattern)

- 关键问题：对于所有 $1 \le i \le len(P)$，求 $max_{ps}(P[:i])$
- 求 $max_{ps}(P[:i+1])$?
- j = len(x) = $max_{ps}(P[:i])$
- 如果 P[i] == P[j]，则 $max_{ps}(P[:i+1])$=j+1 前缀后缀各拓展1
- 否则 j = $max_{ps}(P[:j])$ 继续，新的 j=len(y)
- 如果最后 j=0, $max_{ps}(P[:i+1])$ 也等于 0

y是x的最小子序列，
规模缩小再进行比较



Example: compute for 'p' below:

p	a	b	a	b	a	c	a

Initially: ret[0] = 0
j = 0

i	0	1	2	3	4	5	6
p	a	b	a	b	a	c	a
ret	0	0					

Step 1: i = 1, j=0
ret[1] = 0

i	0	1	2	3	4	5	6
p	a	b	a	b	a	c	a
ret	0	0	1				

Step 2: i = 2, j = 0,
ret[2] = j+1=1

i	0	1	2	3	4	5	6
p	a	b	a	b	a	c	a
ret	0	0	1	2			

Step 3: i = 3, j = 1
ret[3] = j+1=2

Example

Step 4: i = 4, j=2
ret[4] =j+1= 3

i	0	1	2	3	4	5	6
p	a	b	a	b	a	c	a
ret	0	0	1	2	3		

Step 5: i = 5, j = 3,
j=ret[j-1]=ret[2]=0
ret[5] =0

i	0	1	2	3	4	5	6
p	a	b	a	b	a	c	a
ret	0	0	1	2	3	0	

Step 6: i = 6, j = 0
ret[6] = j+1=1

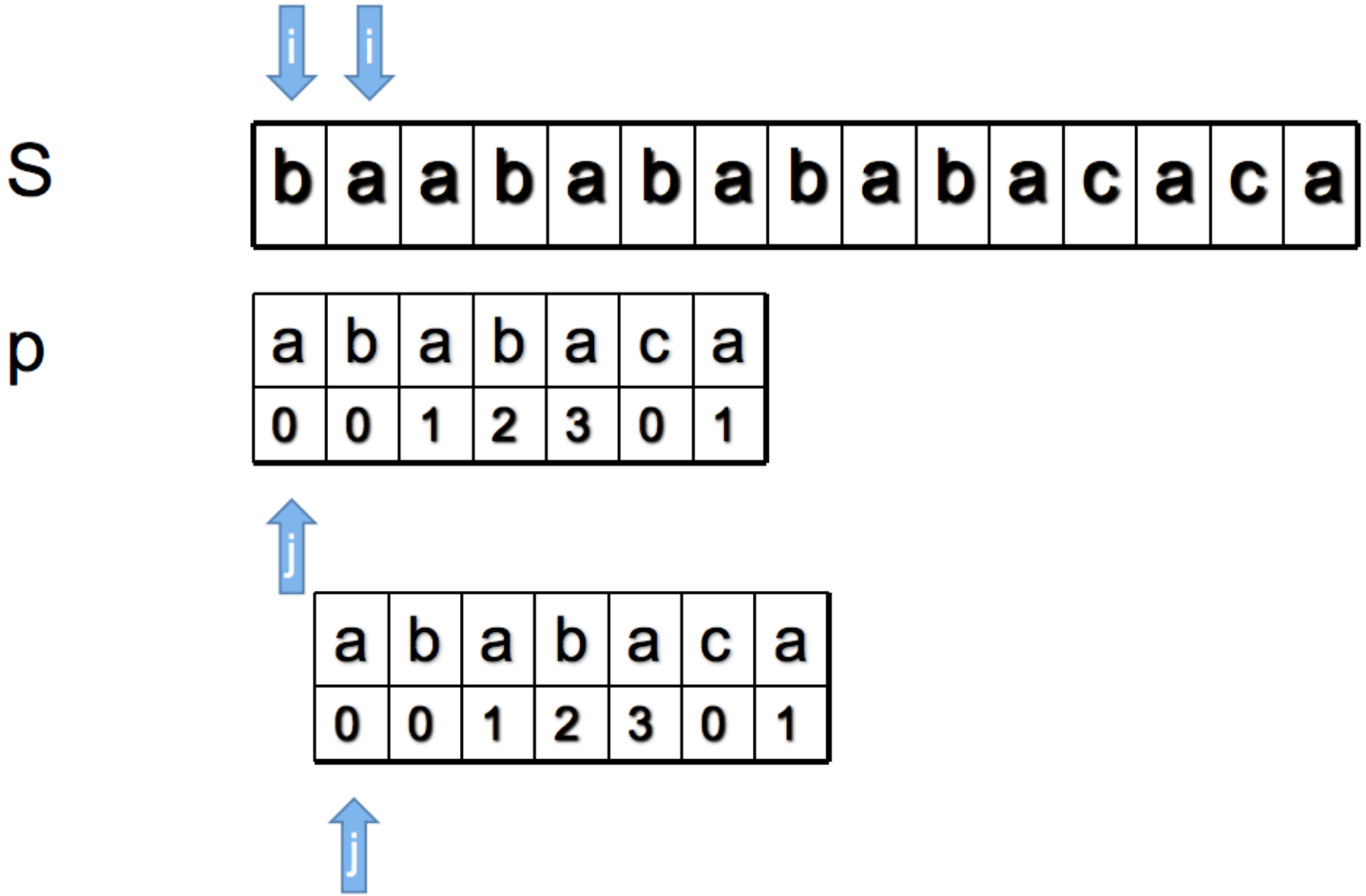
i	0	1	2	3	4	5	6
p	a	b	a	b	a	c	a
ret	0	0	1	2	3	0	1

KMP Main Program

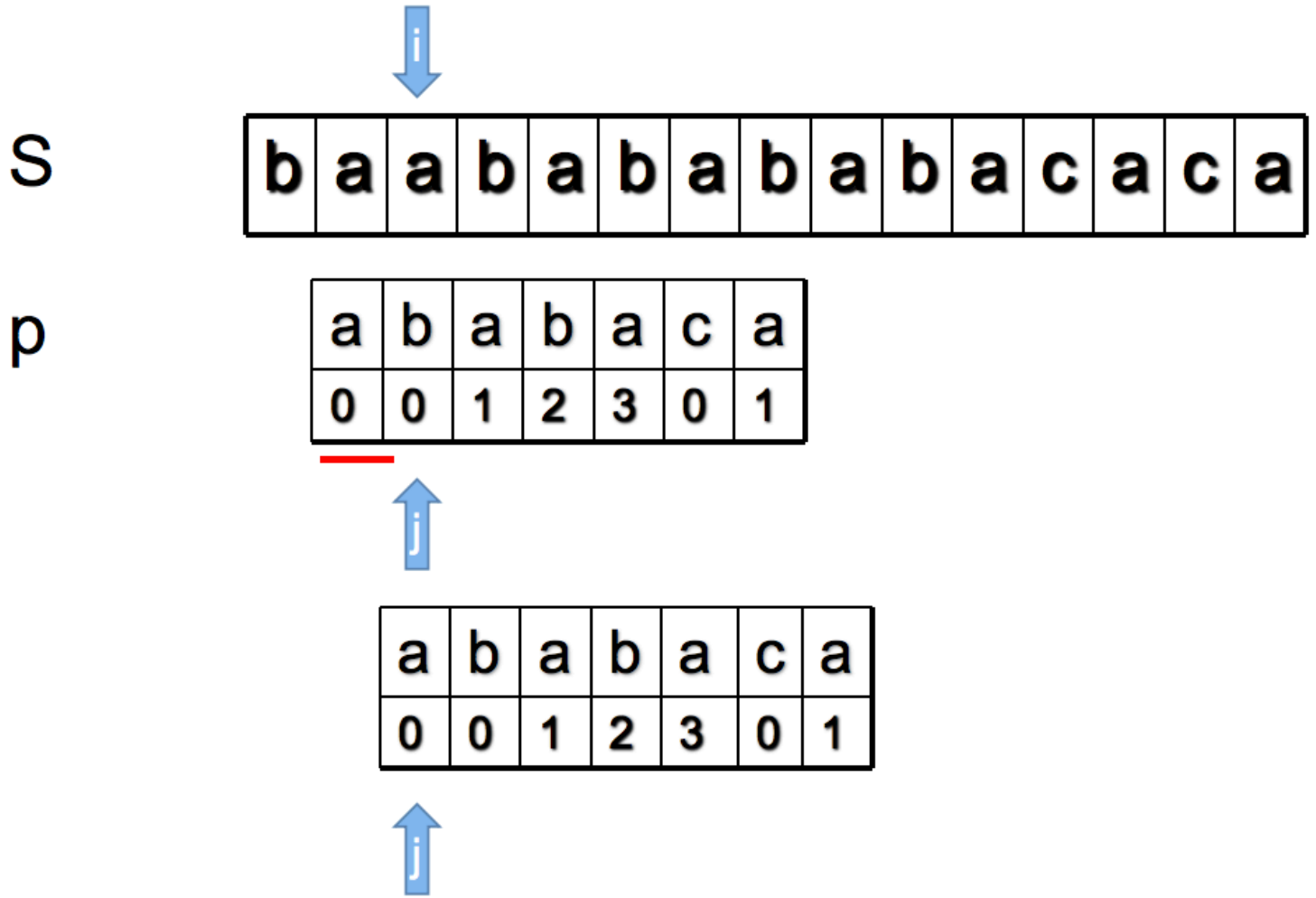
- i 是 T 的比较位置
- J 是 P 的比较位置
- 每次比较失败，j 就跳到 partial[j-1]

```
def search(self, T, P):
    '''
    KMP search main algorithm: String -> String -> [Int]
    Return all the matching position of pattern string P in S
    '''
    partial, ret, j = self.partial(P), [], 0

    for i in range(len(T)):
        while j > 0 and T[i] != P[j]:
            j = partial[j - 1]
        if T[i] == P[j]: j += 1
        if j == len(P):
            ret.append(i - (j - 1))
            j = partial[j - 1]

    print(ret)
    return ret

kmp=KMP()
kmp.search('bacbababababacaab','aaabaca')
```

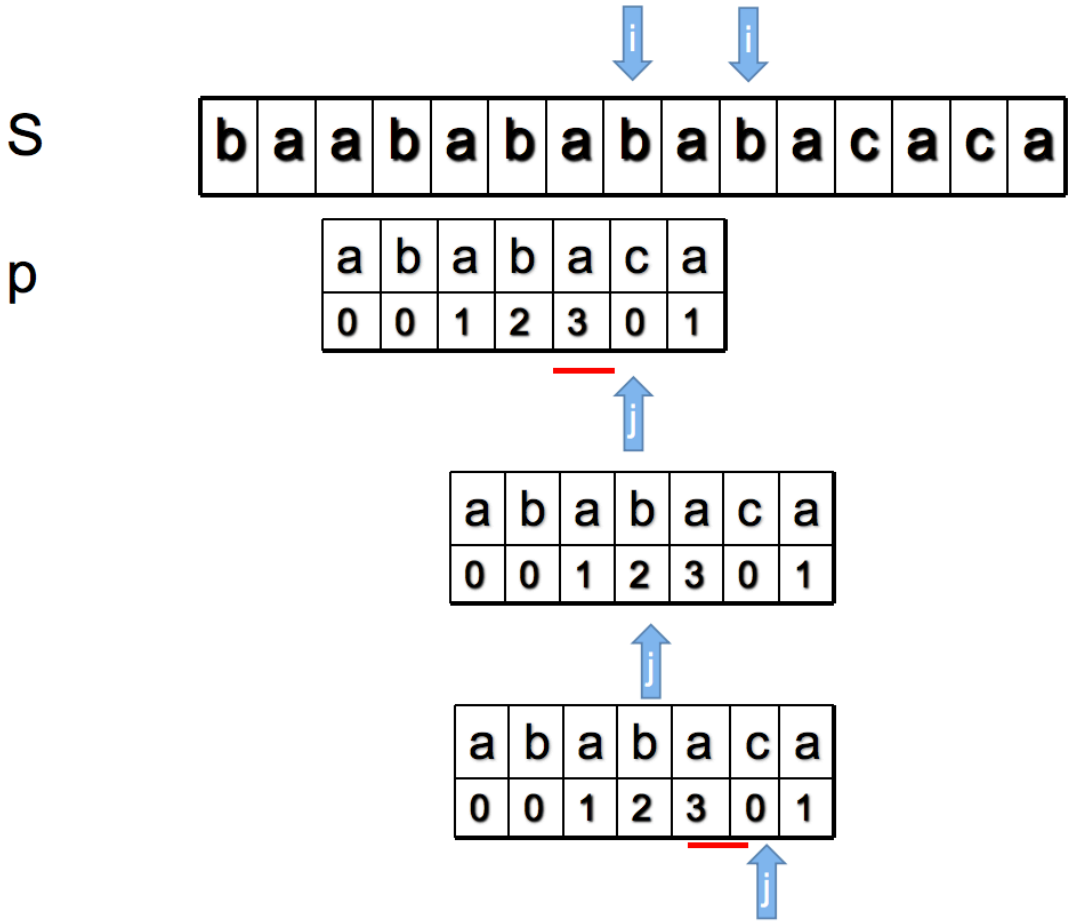
示例



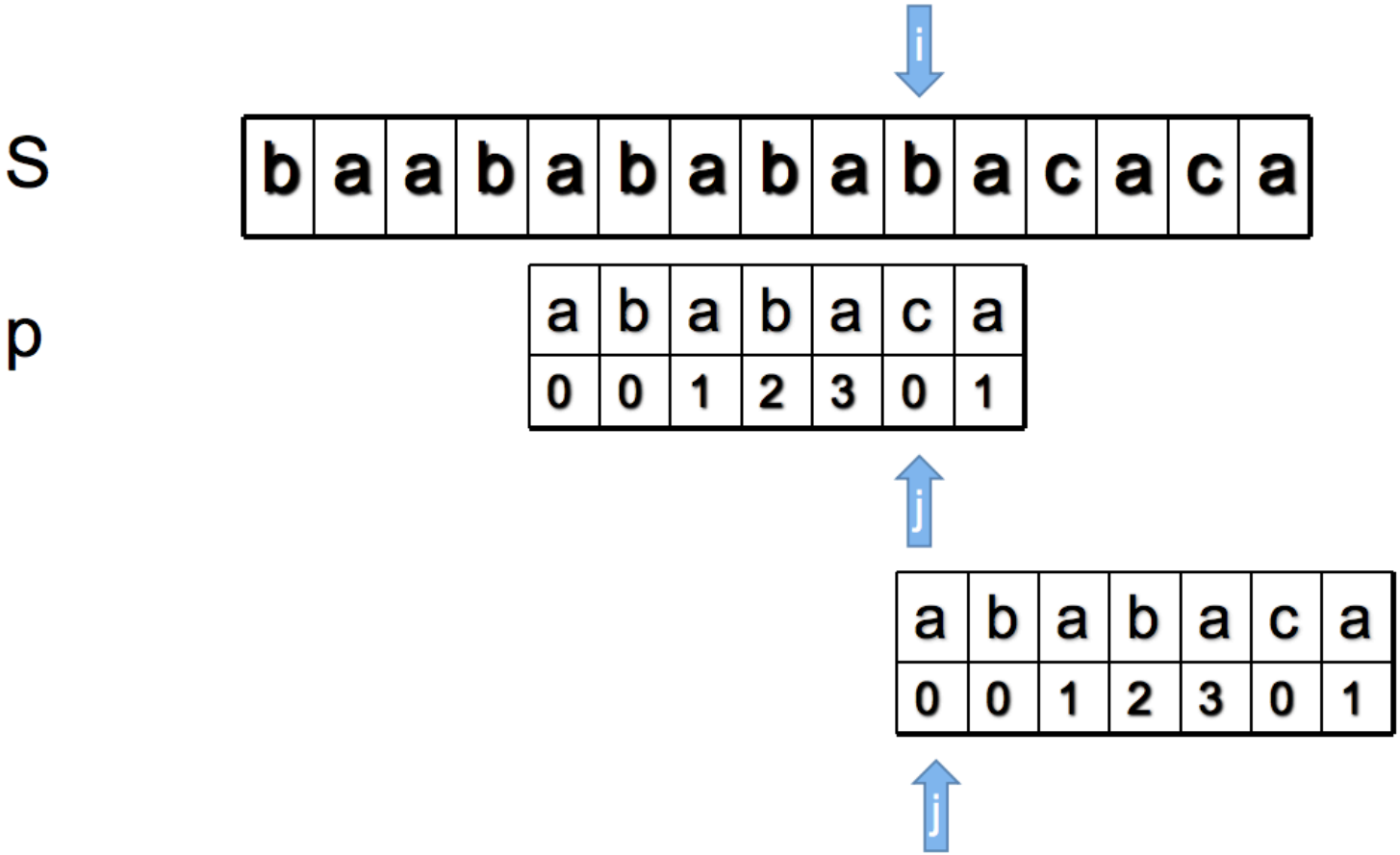
示例



示例


S
b a a b a b a b a b a c a c a
i
i
p
a b a b a c a
0 0 1 2 3 0 1
j
a b a b a c a
0 0 1 2 3 0 1
j
a b a b a c a
0 0 1 2 3 0 1
j

示例



算法复杂度分析

```python
class KMP:
    def partial(self, pattern):
        """ Calculate partial match table: String -> [Int]"""
        ret = [0]

        for i in range(1, len(pattern)):
            j = ret[i - 1]
            while j > 0 and pattern[j] != pattern[i]:
                j = ret[j - 1]
            ret.append(j + 1 if pattern[j] == pattern[i] else j)

        print(ret)
        return ret
```

- 扫描 P 一遍，$O(m)$
- 算法复杂度大致是 $O(m+n)$


```python
class KMP:
    def partial(self, pattern):
        """ Calculate partial match table: String -> [Int]"""
        ret = [0]

        for i in range(1, len(pattern)):
            j = ret[i - 1]
            while j > 0 and pattern[j] != pattern[i]:
                j = ret[j - 1]
            ret.append(j + 1 if pattern[j] == pattern[i] else j)

        print(ret)
        return ret
```


- 扫描 T 一遍，$O(n)$