数据结构与算法（Python）-03+/KMP

刘云淮 Yunhuai.liu@pku.edu.cn

http://www.yunhuai.net/DSA2023/CoursePage/DSA2023.html

北京大学计算机学院

The Problem of String Matching

Given a string ‘S’, the problem of string matching deals with finding whether a pattern ‘p’ occurs in ‘S’ and if ‘p’ does occur then returning position in ‘S’ where ‘p’ occurs.

An O(mn) Algorithm

Compare S and P one by one

If mismatch, then move to S’s next and restart from P’s first

until found

String S

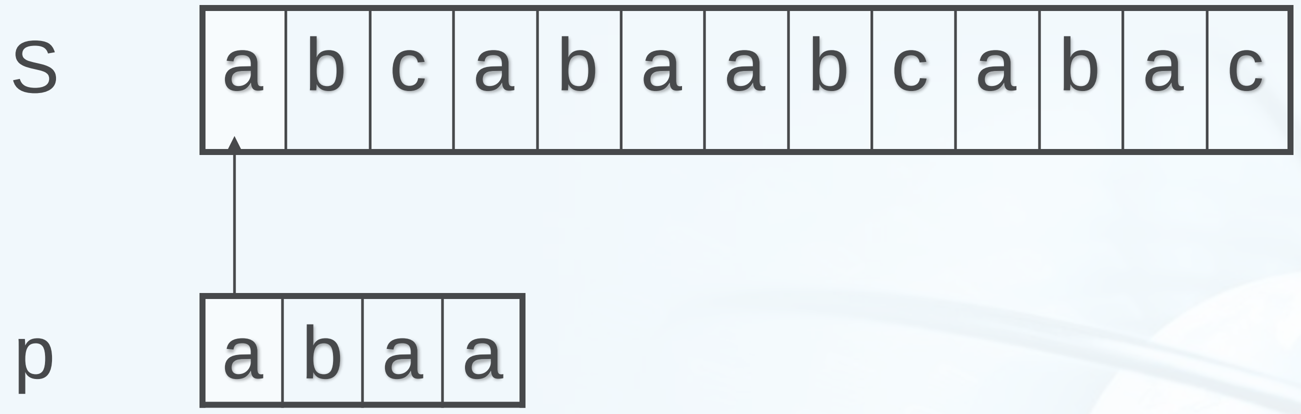
a	b	c	a	b	a	a	b	c	a	b	a	c

Pattern p

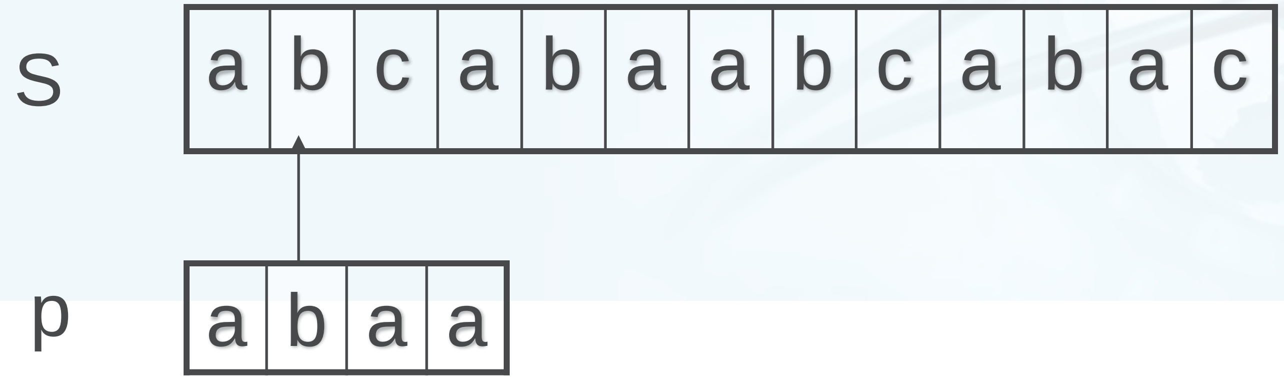
a	b	a	a

How O(mn) Works

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

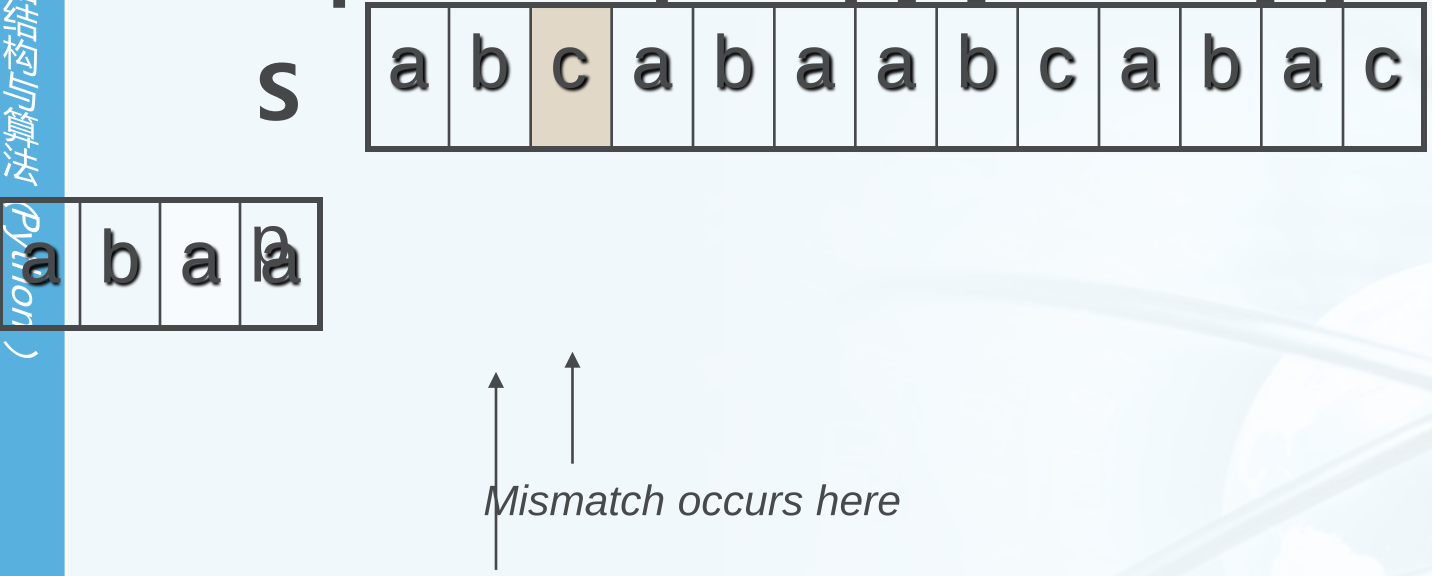


Step 2: compare p[2] with S[2]



How O(mn) Works

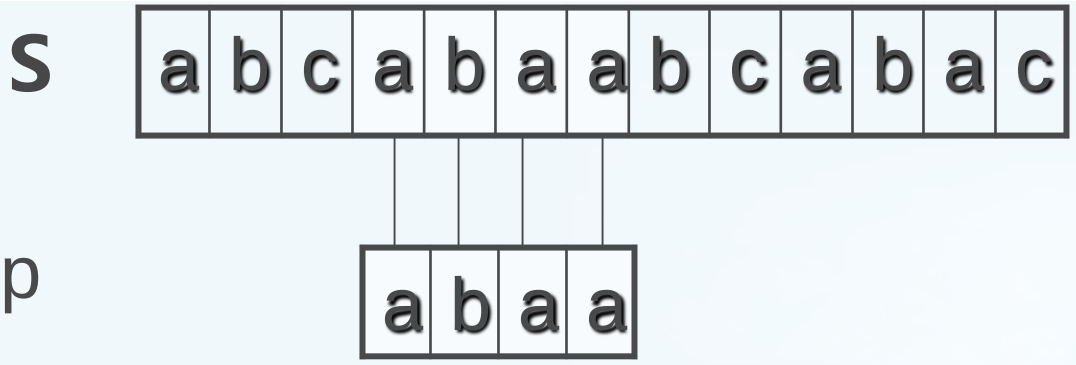
Step 3: compare p[3] with S[3]



Mismatch occurs here

Restart comparison from here

How O(mn) Works



Finally, a match would be found after shifting ‘p’ three times to the right side.

Each S’s element, we need m comparisons in the worst case. There are n elements in S, and therefore the worst running time is O(mn).

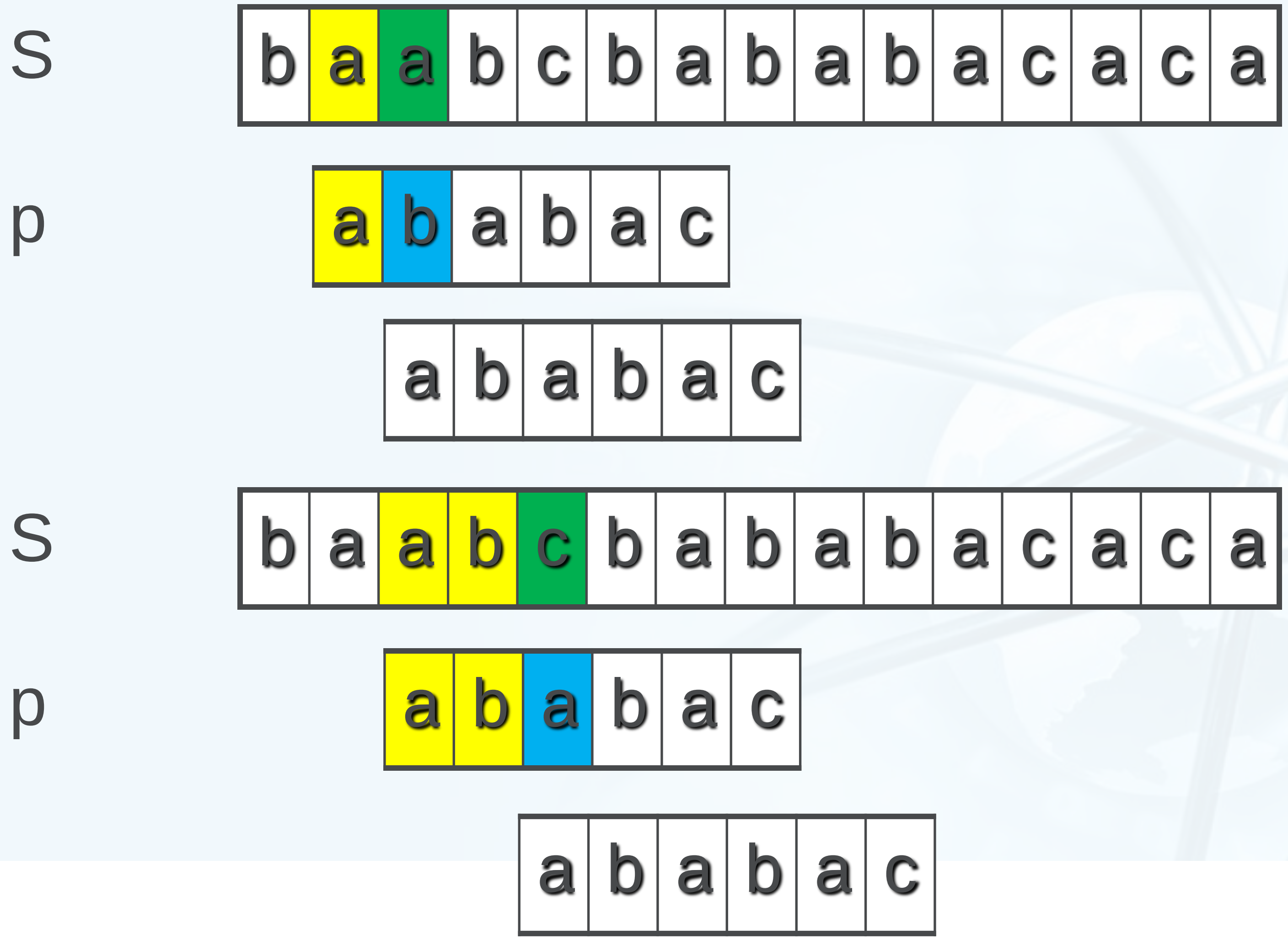
The Knuth-Morris-Pratt Algorithm

Knuth, Morris and Pratt proposed a linear time algorithm for the string matching problem.

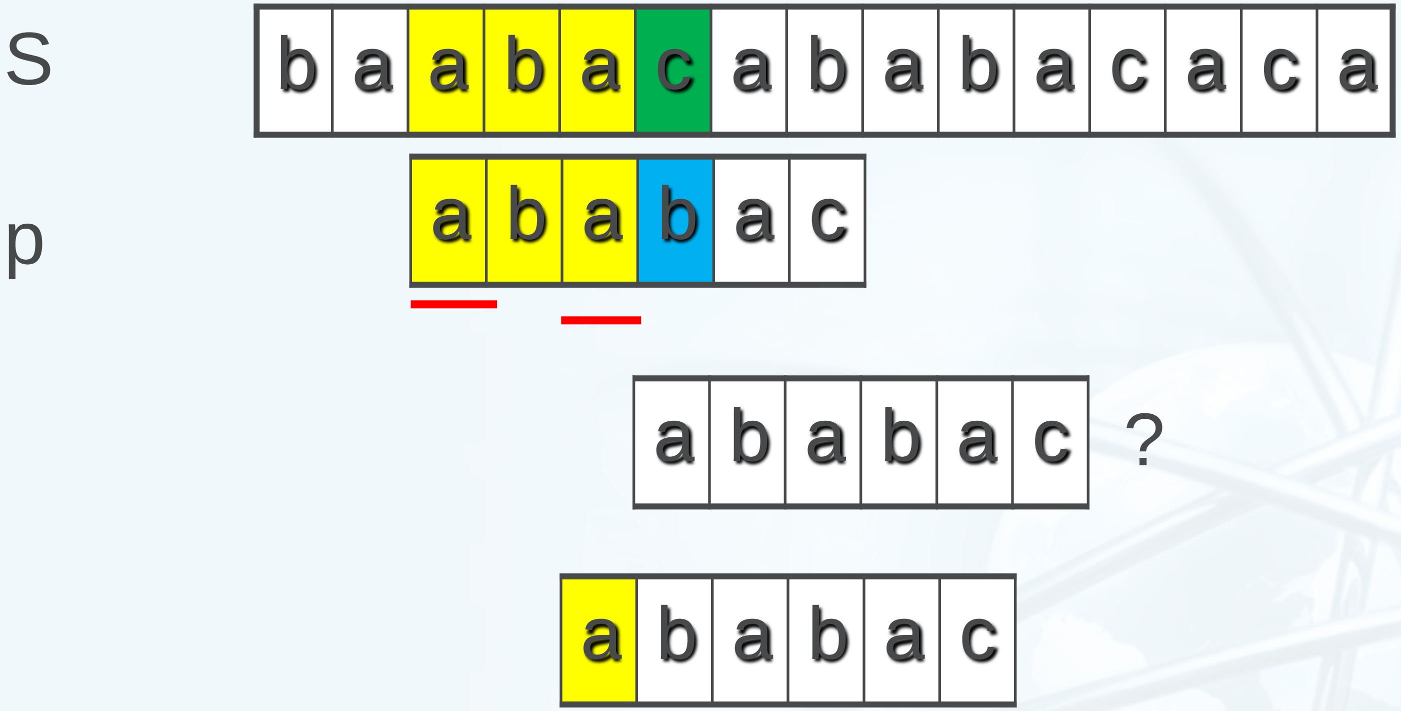
A matching time of O(n+m) is achieved by avoiding comparisons with elements of ‘S’ that have previously been involved in comparison with some element of the pattern ‘p’ to be matched. i.e., backtracking on the string ‘S’ never occurs

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

Observations



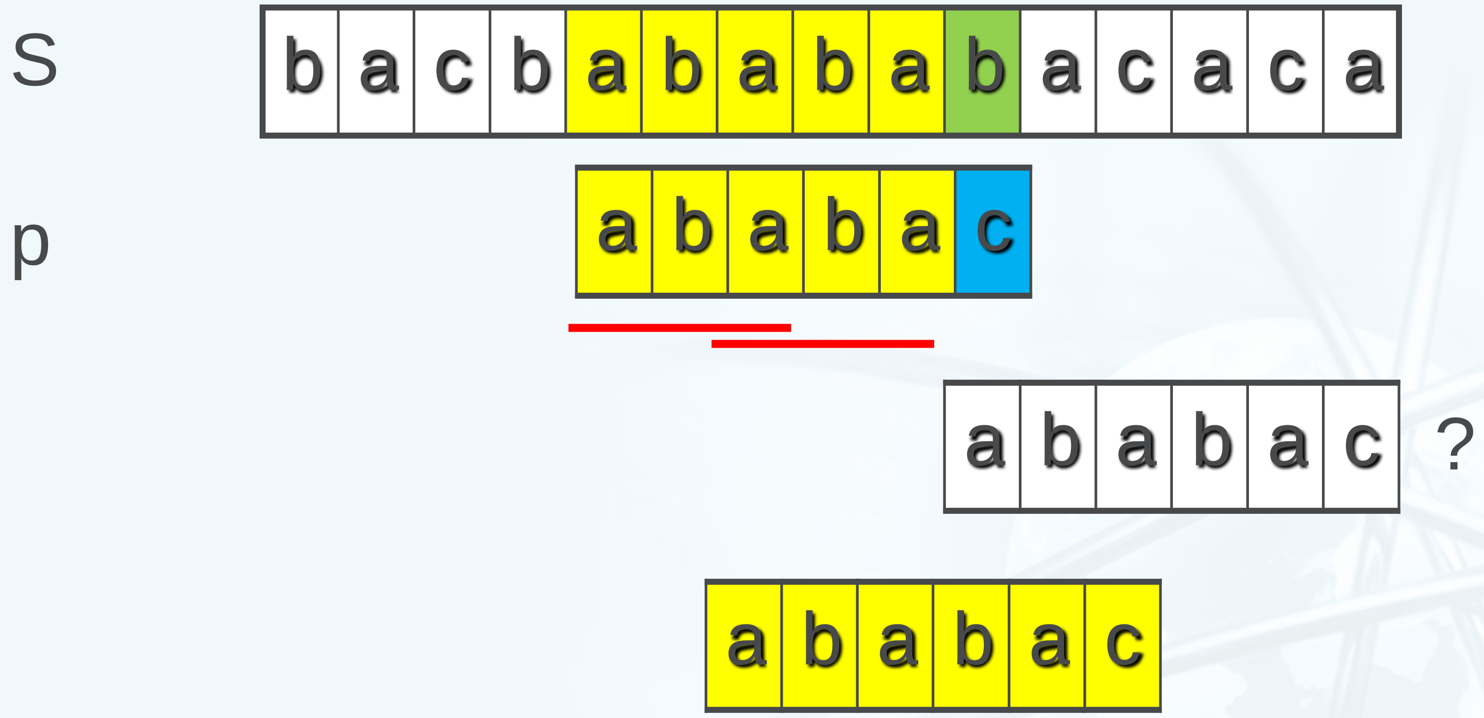
Observations



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

More Observations



p长成什么样子

关键是有多少前缀的和后缀的是相同的

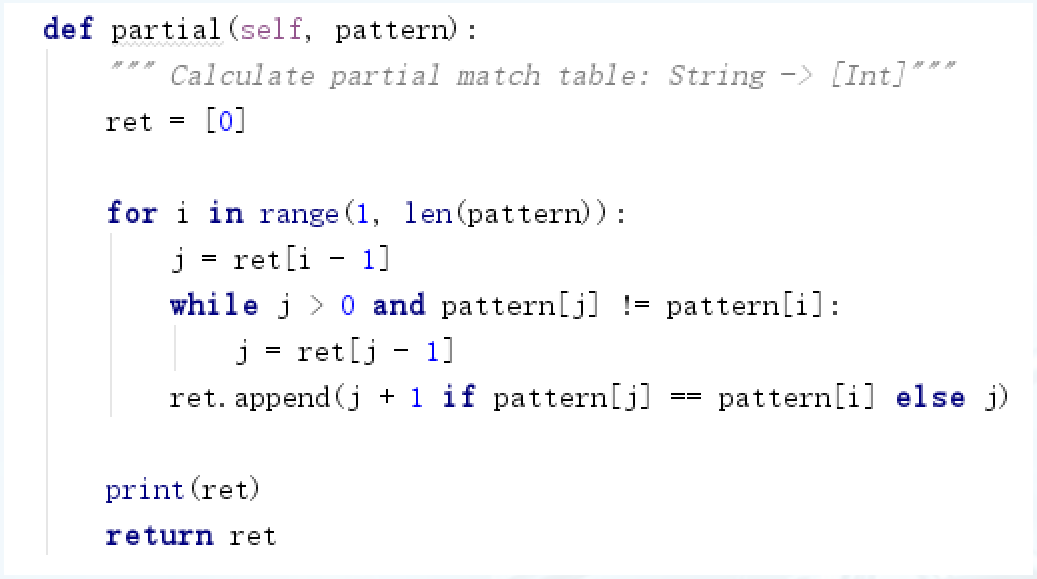
Much More Observations

- 取决于p中任何一位位置看过去，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）

p	a						0
	a	b					0
	a	b	a				1
	a	b	a	b			2
	a	b	a	b	a		3
	a	b	a	b	a	c	0

定义函数partial(prefix, next)

- Pattern , p
- i 是p的索引
- j记录到前一个元素，最长的match
- j=ret[i-1]， 如 果 不 match，则往回寻找可能的最长match，直到找到match, 或者j=0
- 如果match, j=j+1
- 如果不match, j=0



```
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
```

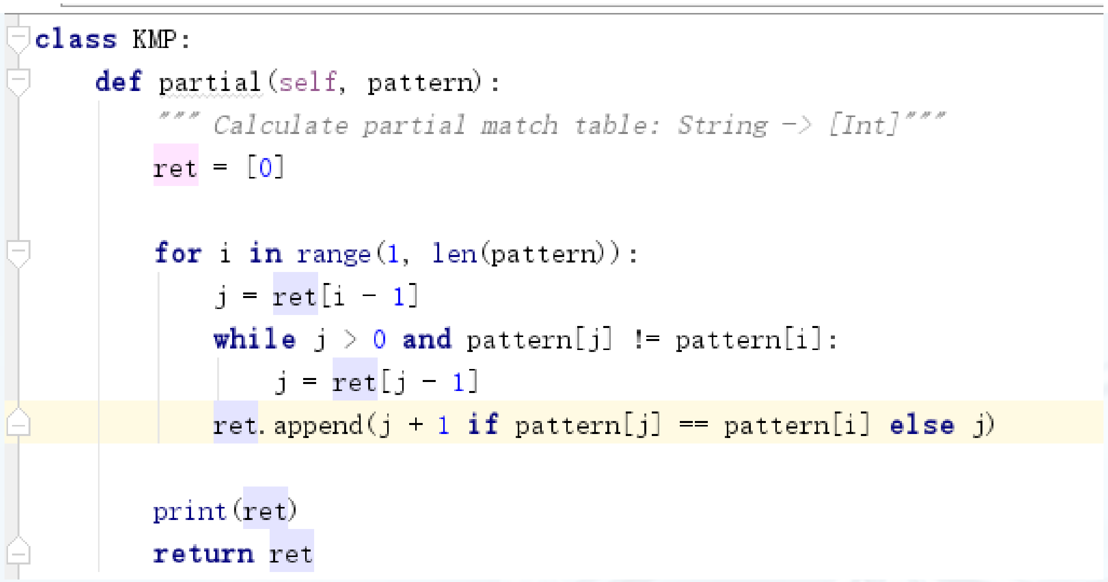
定义函数partial(prefix, next)

- The partial function, ret

The partial function, ret for a pattern encapsulates knowledge about how the pattern matches against shifts of itself. This information can be used to avoid useless shifts of the pattern ‘p’. In other words, this enables avoiding backtracking on the string ‘S’.

定义函数partial(prefix, next)

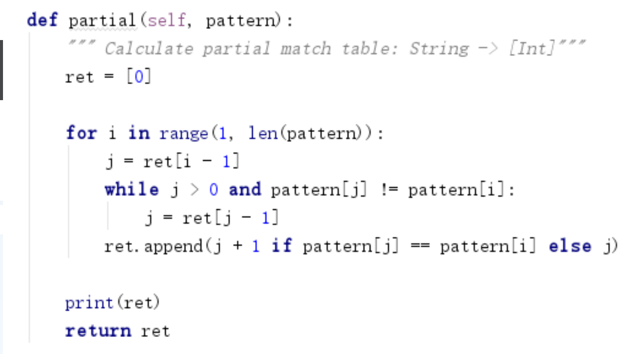
- Pattern , p
- i 是p的索引
- j记录到前一个元素，最长的match
- j=ret[i-1], 如果不match，则往回寻找可能的最长match，直到找到match, 或者j=0
- 如果match, j=j+1
- 如果不match, j=0



```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
```

Example: compute for ‘p’ bel

p	a	b	a	b	a	c	a



```
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
```

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					

<u>Step 1:</u> i = 1, j=0
ret[1] = 0

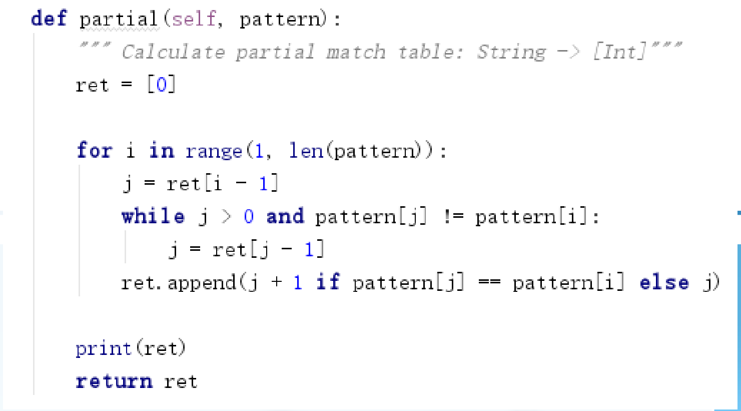
i	0	1	2	3	4	5	6
p	a	b	a	b	a	c	a
ret	0	0	1				

<u>Step 2:</u> 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			

<u>Step 3:</u> i = 3, j = 1
ret[3] = j+1=2

Example



```
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
```

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

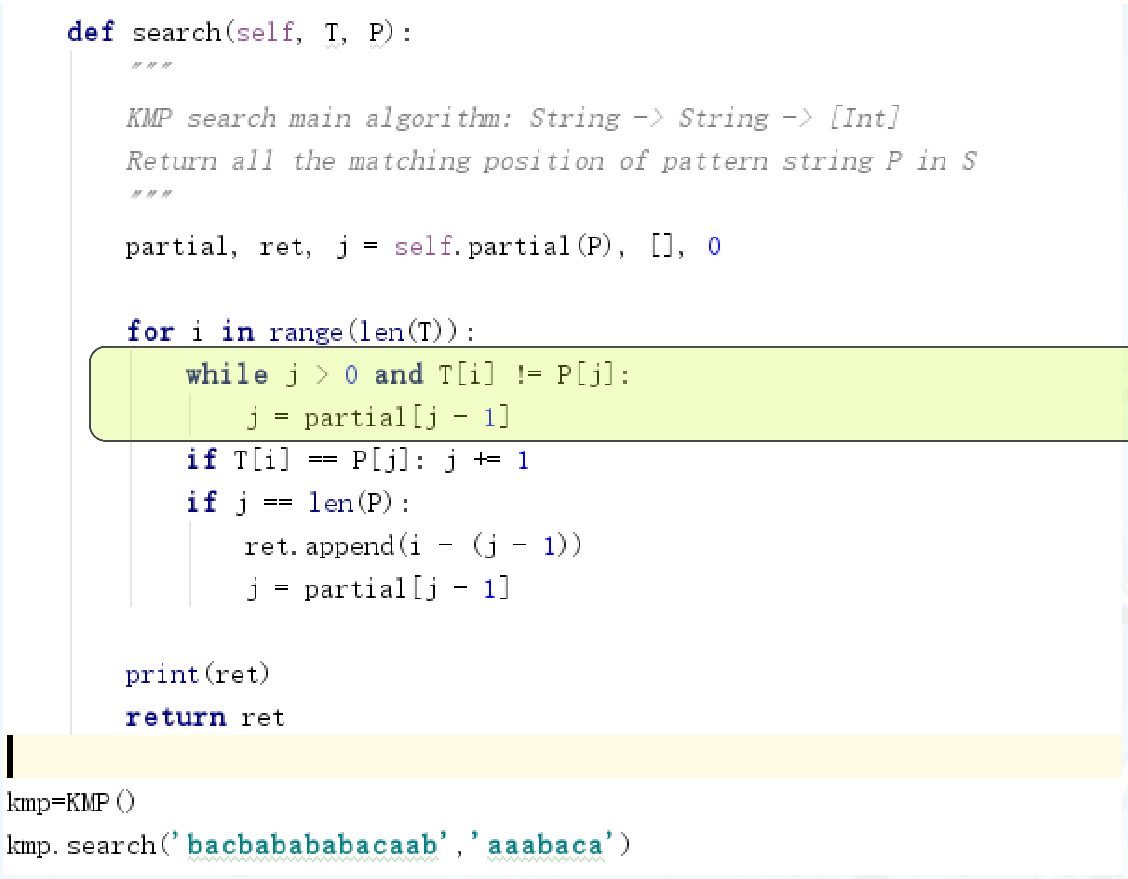
i	0	1	2	3	4	5	6
p	a	b	a	b	a	c	a
ret	0	0	1	2	3	0	

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

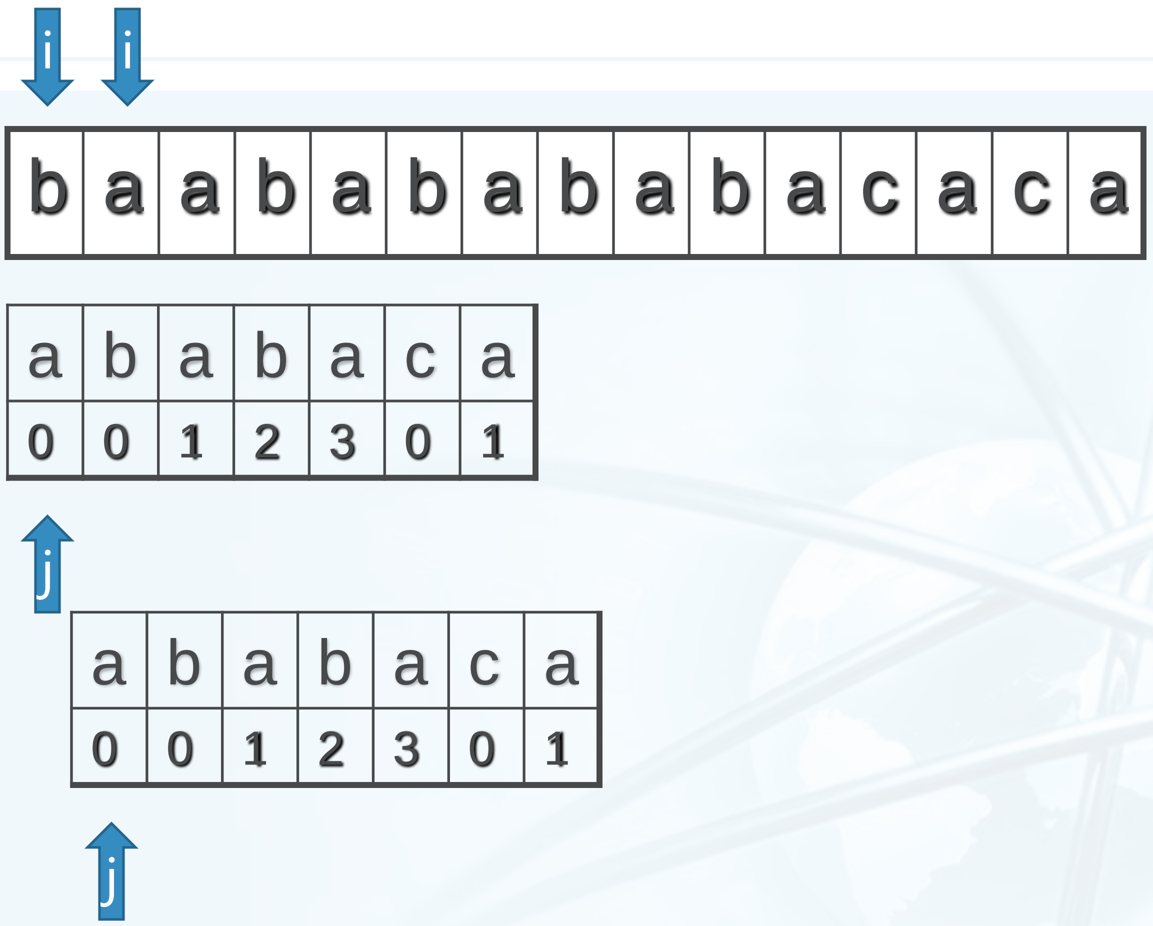
KMP Main Program

- 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]

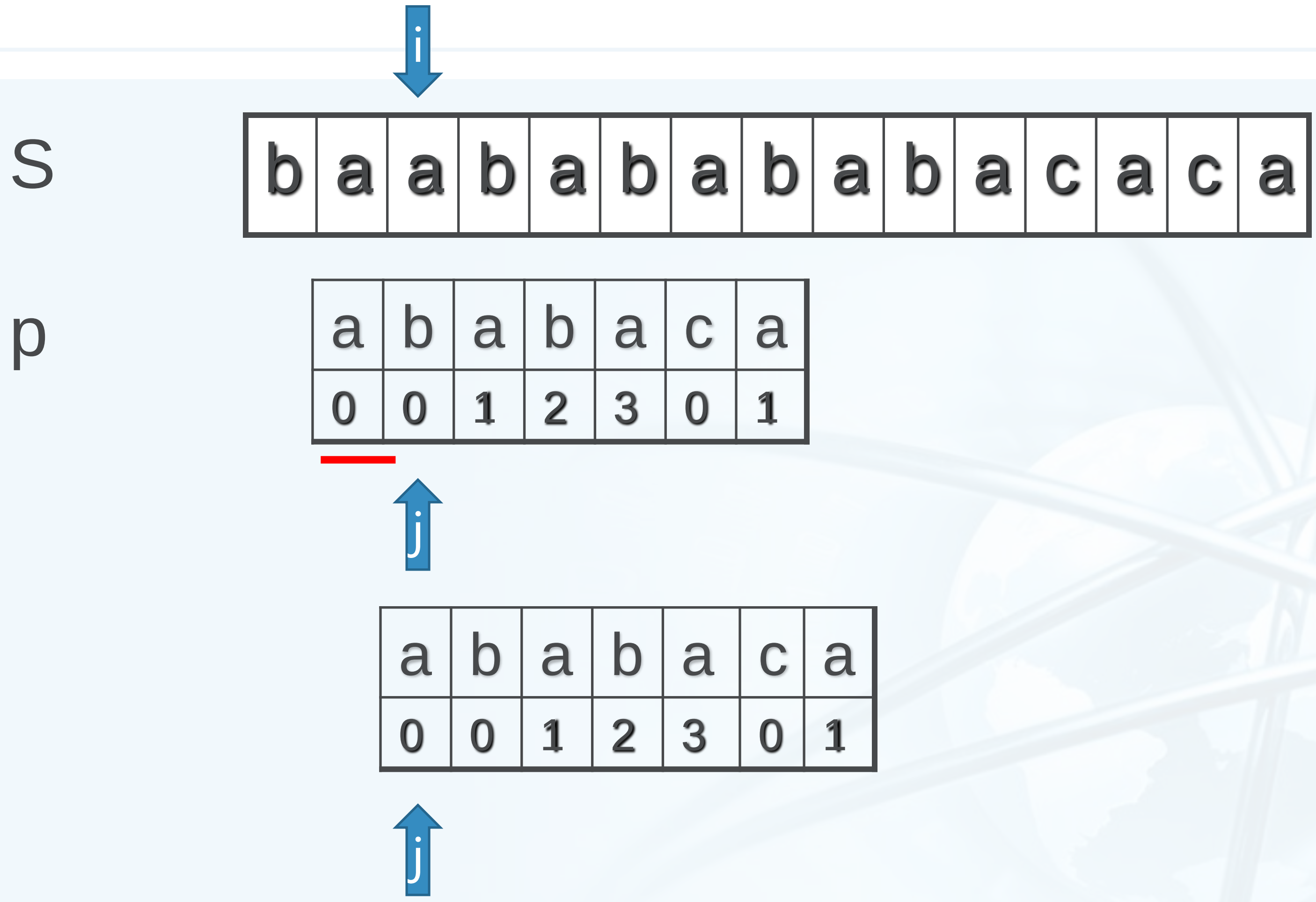


```
def search(self, T, P):
    """
    KMP search main algorithm: String -> String -> [Int]
    Return all the matching position of pattern string P in S
    """
    partial, ret, j = self.partial(P), [], 0

    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('bacbababacaab','aaabaca')
```

示例



示例

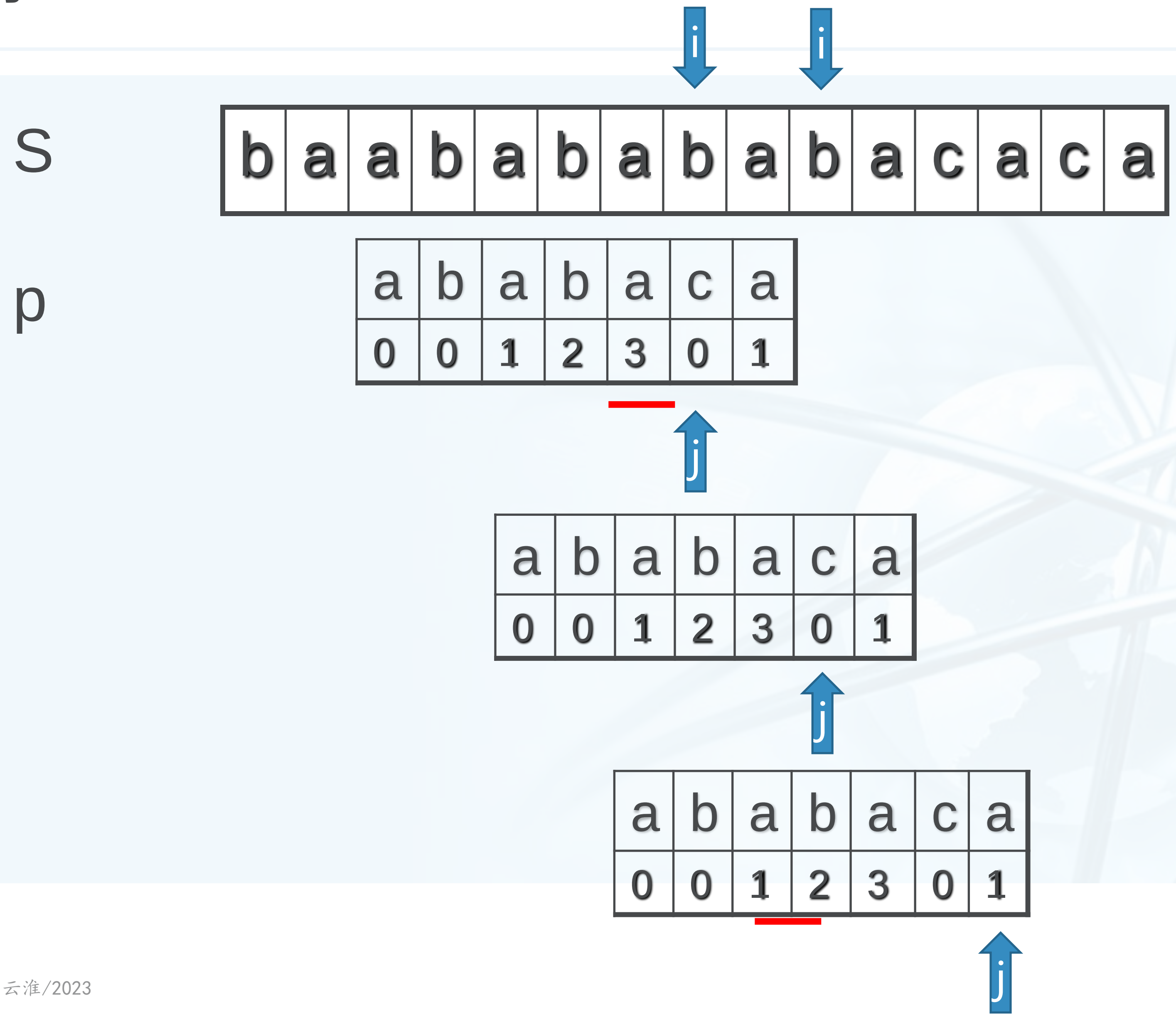


b	a	a	b	a	b	a	b	a	b	a	c	a	c	a

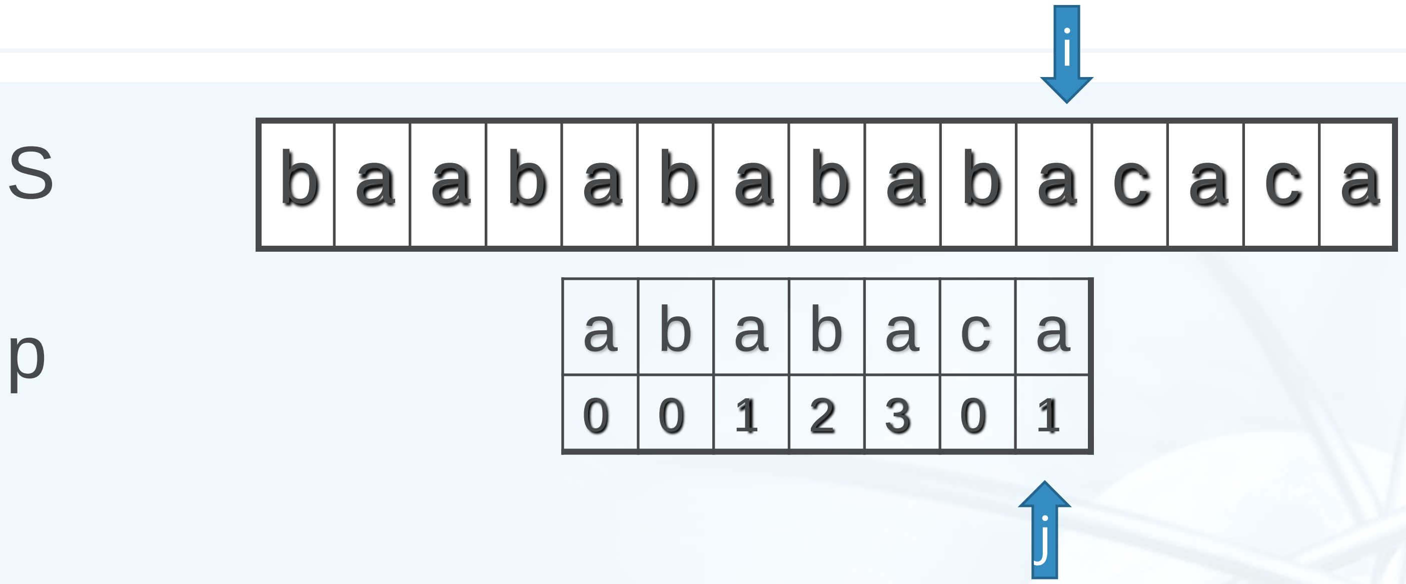
a	b	a	b	a	c	a
0	0	1	2	3	0	1

a	b	a	b	a	c	a
0	0	1	2	3	0	1

示例


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

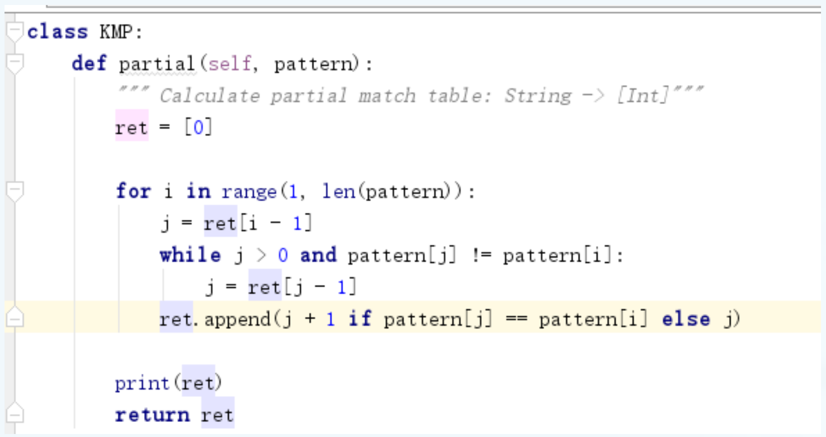
示例



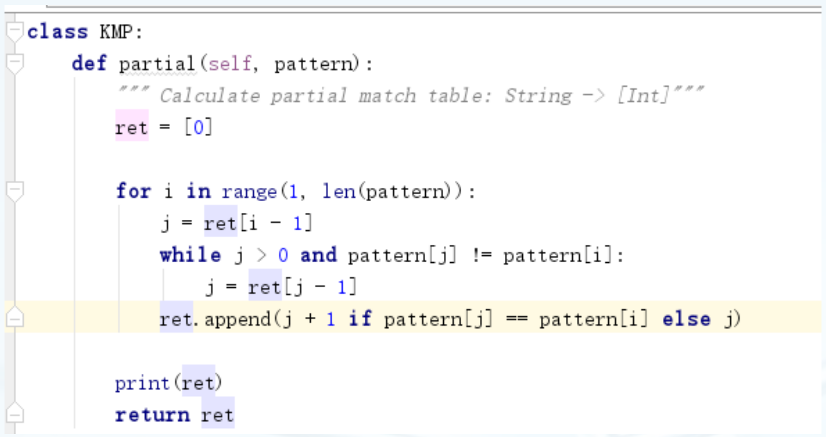
b	a	a	b	a	b	a	b	a	b	a	c	a	c	a

a	b	a	b	a	c	a
0	0	1	2	3	0	1

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
```



```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)
- 扫描T一遍，O(n)
- 算法复杂度是O(m+n)