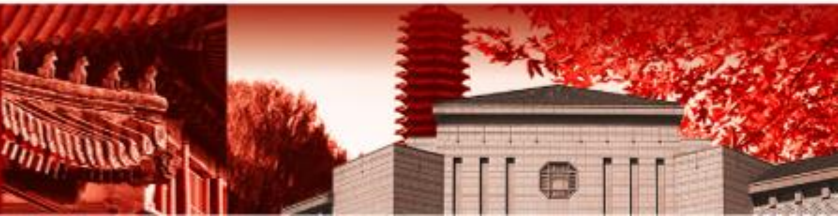
数据结构与算法B

03 – 字符串



目录

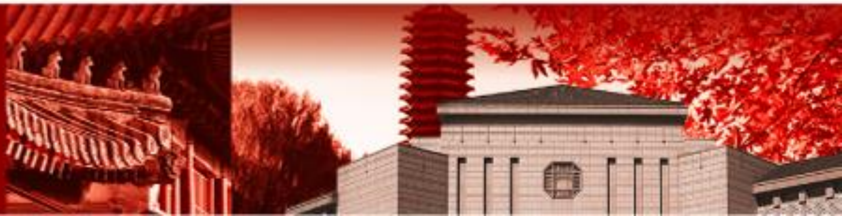
- 3.1 字符串基本概念
- 3.2 Python中的字符串
- 3.3 模式匹配算法
  - 朴素模式匹配算法
  - 无回溯模式匹配(KMP)算法

3.1 字符串的基本概念

字符串示例

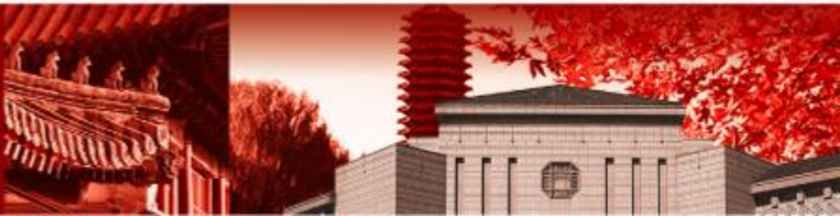
- s1="123"
- s2="Data Structure and Algorithm"
- s3="Hello World"
- s4="BB "
- s5=""

字符串：简称为串，是零个或多个字符组成的有限序列。一般记为：s="$s_0s_1…s_{n-1}$"（n≥1），s为串名，每个字符s_i (0 <= i <= n-1)可以是字母、数字或其它字符



术语表

- **串的长度**：字符串中字符个数
- **空串**：长度为零的字符串，记为s = “”
  - 注意与空格串 “ ” 的区别
- **子序列**：字符串s1中，任意个字符按顺序组成的序列s2称为s1的子序列
  - s1=“Hello world”, s2=“Helo word”
- **子串与主串**：字符串s1中任意个连续字符组成的序列s2称为s1的子串，称s1为s2的主串。
  - s1=“Hello world”, s2=“Hello”
  - 空串是任意串的子串; 除s本身之外，s的其他子串称为s的真子串。

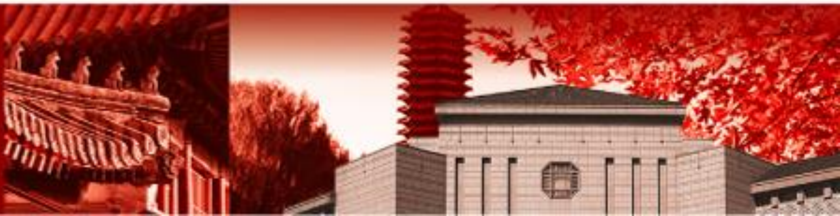


术语表

- **字符在串中的位置**：该字符在串中第一次出现时的位置。
- **子串在主串中的位置**：该子串在串中第一次出现时，第一个字符的位置。
- **两个字符串相等的充分必要条件**：长度相等，且对应位置上字符相同。

字符串是一种特殊的线性结构

- 字符串与一般线性表的区别与联系
  - 串的数据对象约束为字符集，其每个结点包含一个字符；
  - 线性表的基本操作大多以“单个元素”为操作对象，而串的基本操作通常以“串的整体”作为操作对象；
  - 线性表的存储方法同样适用于字符串，在选择存储结构时，应根据不同情况选择合适的存储表示。



3.2 Python中的字符串

Python中的字符串

- Python中，str类型是不可变类型
  - 字符串可以用单引号 '、双引号 " 或三引号 '''/""" 创建。

    例如：s1 = 'Hello‘
    s2 = "World"
    s3 = """This is a multi-line
    string"""
  - 不可变类型的变量，一经创建，不可修改其值
  - 回忆：同样作为容器的类型，list是可变类型，tuple则是不可变类型
- 那么，Python如何实现各种字符串操作？
  - 一些操作不需要修改字符串：切片（如s[1:4]）、查找子串（如s.find("is")
  - 对于修改操作，通过创建新的字符串来实现对字符串的“修改”
  - 最常见的拼接操作，即创建一个新的字符串
    - 例如，S = “Python ” + “is ” + “awesome”
    - 拼接大量字符串时，造成性能下降
    - 使用join运算：S = “ ”.join([“Python”, “is”, “awesome”])

Python中的字符串

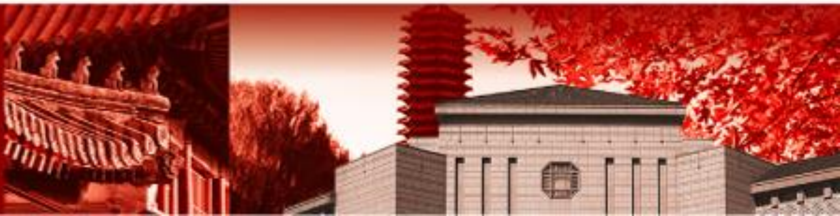
- Python中str类型的常用方法
  - __len__ 返回串的长度
  - __add__ 两个字符串的拼接
  - __eq__, __gt__, __lt__ 字符串的相等/大于/小于运算（字典序）
  - split 以指定分隔符（默认空格）分割字符串
  - join 将序列中的元素以指定字符串连接
  - strip 移除字符串两侧的空格
  - find 在字符串中检索子串

示例1

```
>>> s2 = "Alice, Bob, Carol, Dave, Eve"
>>> s2.split(",")
['Alice', ' Bob', ' Carol', ' Dave', ' Eve’]

>>> [name.strip() for name in s2.split(",")]
['Alice', 'Bob', 'Carol', 'Dave', 'Eve’]

>>> ", ".join(["Alice", "Bob", "Carol", "Dave", "Eve"])
'Alice, Bob, Carol, Dave, Eve'
```



示例2

```
>>> s1 = "Find a substring in string s1"
>>> s1.find("substring")
7
>>> s1.find("string s1")
20
>>> s1.find("string s2")
-1
```

- 如何高效实现find操作？
- 本节课来重点探讨

3.3 模式匹配算法

问题定义

- 模式匹配的定义
  - 设有两个串 $t = t_0t_1\ldots t_{n-1}$ (**target**) 和 $p = p_0p_1\ldots p_{m-1}$ (**pattern**)，其中 $1<m<=n$
  - 在t中找出和p相同的子串。此时，t称为“目标”，而p称为“模式”

```
def match(target: string, pattern: string );
```

  - 匹配成功：t中存在等于p的子串，返回子串在t中的位置
  - 匹配失败：返回一个特定的标志（如-1）。

两种方法

- 模式匹配是一个比较复杂的字符串操作，
  - 用于在大文本（诸如，句子、段落，或书本）中定位（查找）特定的模式
  - 对于大多数的算法而言，匹配的主要考虑在于其速度和效率

- 下面的讨论是基于字符串的顺序存储结构的算法
  - 朴素的模式匹配(Brute Force)方法
  - 无回溯的模式匹配Knuth-Morris-Pratt (KMP)方法

朴素的模式匹配思想

- 用P中的字符依次与T中的字符比较：
  - 如果P[0:m] == T[0:m]，则匹配成功;
  - 否则，将p右移一个字符，用p中字符从头开始与t中字符依次比较。
  - 如此反复执行，直到下面两种情况之一：
    - 到达某步时，P[0:m] == T[i: i+m]，匹配成功；
    - 将P移到无法与T继续比较为止，则匹配失败

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
T	a	b	c	a	a	b	a	b	c
P	a	b	a	b					
	p_0	p_1	p_2	p_3					

朴素的模式匹配算法

```python
def BruteForce(pattern, target):
    length_p, length_t = len(pattern), len(target)
    s = i = j = 0
    while i < length_p and j < length_t:
        if pattern[i] == target[j]:
            i, j = i+1, j+1
        else:
            s, i = s+1, 0
            if s > length_t - length_p:
                break
            j = s
    if i == length_p:
        return s
    else:
        return -1
```

i, j表示pattern, target的比较位置，s表示当前比较中target的匹配起点，即匹配成功后待返回的值

失配后，将target的匹配起点右移一位继续比较

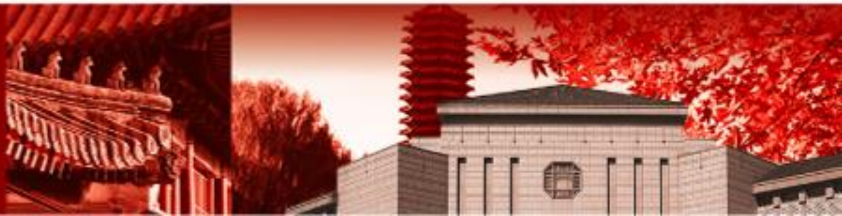
若target剩余字符数小于模式串长度，则匹配失败

模式串每个字符都匹配上了

算法时间效率分析

target串长度为n，pattern串长度为m

- 匹配失败
  - **最坏情况：**每趟匹配皆在最后一个字符不等，且有n-m+1趟匹配(每趟比较m个字符），共比较m*(n-m+1)次。
  - 通常m<=n，因此最坏时间复杂度O(n*m)
  - **最好情况：**比较n-m+1次 [每趟只比较第一个字符]
- 匹配成功
  - **最好情况：**m次比较
  - **最坏情况：**与匹配失败的最坏情况相同，即比较m*(n-m+1)次。
- 因此，朴素模式匹配算法的时间复杂度为O(m*n)



进一步的分析

- 朴素模式匹配算法简单，容易理解，但效率不高。
  - 主要原因是：一旦比较不等，将p右移一个字符后，需要从p_0开始重新进行比较
  - 是否可能减少或者避免回溯？

分析这个例子中的回溯：

- 已经完成了2位匹配：p0=t0, p1=t1, p2 ≠ t2
- p0≠p1可以推出p0≠t1，所以p右移一位必定匹配失败
- p0=p2可以推出p0≠t2，所以p右移二位必定匹配失败
- 因此，可以由第1趟匹配直接跳过2、3趟匹配进入第4趟匹配，减少了回溯

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
T	a	b	c	a	a	b	a	b	c
P	a	b	a	b					
	p_0	p_1	p_2	p_3					

进一步的分析

- 前例中，一旦P[i]与T[j]比较不等，有：
  - P[0:i] == T[j-i: j]，$P[i] \neq T[j]$
  - 此时，朴素匹配算法将P右移一位
  - 而有时实际上可以将P右移多位，同时保证匹配的正确性。
- 应该将P右移多少位（记为k），继续与T比较？
- D.E.Knuth、J.H.Morris和V.R.Pratt同时发现：
  - k的值仅与P(pattern)的性质有关，而与T(target)无关！

KMP算法

- 前例中，一旦P[i]与T[j]比较不等，有：
  - P[0:i] == T[j-i: j], P[i] ≠ T[j]
- 依照朴素算法右移P，实际上做的比较：

T[0]	**……**	**T[j-i]**	**T[j-i+1]**	**T[j-i+2]**	**……**	**T[j-2]**	**T[j-1]**	**T[j]**	**T[j+1]**

P[0]	**P[1]**	**P[2]**	**……**	**P[i-2]**	**P[i-1]**	**P[i]**

T[0]	**……**	**P[0]**	**P[1]**	**P[2]**	**……**	**P[i-2]**	**P[i-1]**	**T[j]**	**T[j+1]**

KMP算法

- 前例中，一旦P[i]与T[j]比较不等，有：
  - P[0:i] == T[j-i: j], P[i] ≠ T[j]
- 依照朴素算法右移P，实际上做的比较：
  - 尝试将P[0:i]的前缀与后缀进行比较！
  - 右移i-k位时，就是在比较长为 k 的前缀与后缀(下图举例k=2)

			……	**T[j-k]**	**T[j-1]**	**T[j]**	**T[j+1]**

P[0]	**P[1]**	**P[k]**	……	**P[i-k]**	**P[i-1]**	**P[i]**

P[0]	**P[1]**	**P[k]**	……		

将P右移i-k 位

KMP算法

- 前例中，一旦P[i]与T[j]比较不等，有：
  - P[0:i] == T[j-i: j], P[i] ≠ T[j]
- 依照朴素算法右移P，实际上做的比较：
  - 尝试将P[0:i]的前缀与后缀进行比较！
  - 右移i-k位时，就是在比较长为 k 的前缀与后缀
- 如何利用这一点来减少冗余比较？
  - 寻找P的最大长度公共前后缀（记为k）
  - 右移小于i-k位的尝试，都一定失败
    - 因为它在尝试比较长度大于k的前后缀
  - 因此，可以跳过上述尝试，右移i-k位，并直接从P[k], T[j]位置开始比较
  - 达成的效果是，在T的视角来看，完全没有回溯发生！

前后缀指，既是字符串的真前缀，也是字符串的后缀。例如，abab的前后缀是ab

KMP算法：Next数组

- Next数组：Next[i] 表示 P[0: i] 的最长公共前后缀
  - 公共前后缀（或前后缀）需要同时是真前缀和后缀，排除了字符串本身
  - 如何计算?
- 方法：假定已经得到Next[i] = k, 如何计算Next[i+1]?
  - 如果P[k] = P[i]： Next[i+1] = k+1
  - 如果P[k] ≠ P[i]?

P[0]	**……**	**P[k-1]**	**P[k]**	**……**	**P[i-k-1]**	**P[i-k]**	**……**	**P[i-1]**	**P[i]**

KMP算法：Next数组

- 如果P[k] ≠ P[i]，如何计算Next[i+1]?
  - 首先，Next[i+1] < k+1；否则容易说明与Next[i]=k矛盾
  - 性质：若字符串S是P[0: i+1]的公共前后缀（下图第2行表示S），则S去掉最后一位后，构成P[0: i]的公共前后缀（下图第2行标红部分，记S'）
    - 进一步，还构成P[0, k]的公共前后缀
  - 利用P[0: k]的最长公共前后缀（假定已经求出Next[k]=k_1），并比较P[i]与P[k_1]
    - 如果不相等，继续考虑Next[k_1]，依此类推，直到发现公共前后缀

P[0]	**……**	**P[k-1]**	**P[k]**	**……**	**P[i-k-1]**	**P[i-k]**	**……**	**P[i-1]**	**P[i]**

P[0: k_1]	**P[k_1]**		**P[0: k_1]**	**P[i]**

P[0: k_2]	**P[k_2]**		**P[0: k_2]**	**P[i]**

KMP算法

```python
def make_next(pattern):
    Next = [None] * len(pattern)
    i, k = 0, -1
    while i < len(pattern) - 1:
        if i == 0:
            Next[i] = -1
        while k >= 0 and pattern[i] != pattern[k]:
            k = Next[k]
        k += 1
        i += 1
        Next[i] = k
    return Next
```

计算Next[i]，即前i个字符的最长公共前后缀

边界情况，定义为-1

由于所有规模更小的Next[k]都已经求解，如此循环就实现了递归

计算next数组的复杂度分析

```python
def make_next(pattern):
    Next = [None] * len(pattern)
    i, k = 0, -1
    while i < len(pattern) - 1:
        if i == 0:
            Next[i] = -1
        while k >= 0 and pattern[i] != pattern[k]:
            k = Next[k]
        k += 1
        i += 1
        Next[i] = k
    return Next
```

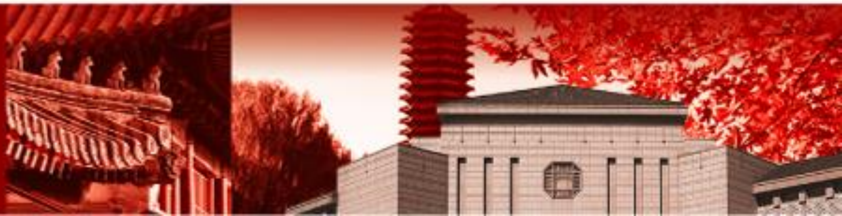
这一层循环O(m)次，m为pattern长度

这一层最多循环多少次，O(m)?

考察程序的整个运行过程中k值的变化

- k += 1 处使k的值增大
- k = Next[k] 处使k的值减小
- 而我们知道，k += 1 运行O(m)次
- 所以 k=Next[k] 运行次数同样为O(m)，无需再关注这一层循环多少次

总结：时间复杂度为O(m)，m为pattern长度



KMP算法

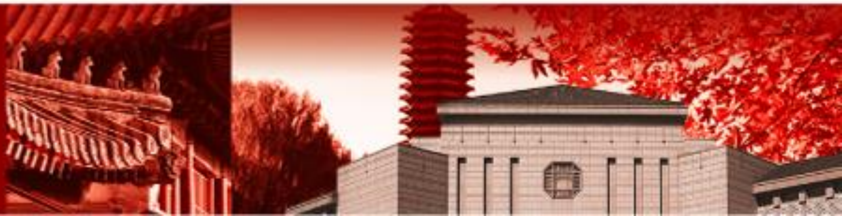
```python
def match(target, pattern, Next):
    i, j = 0, 0
    while i < len(pattern) and j < len(target):
        if pattern[i] == target[j] or i == -1:
            i += 1
            j += 1
        else:
            i = Next[i]
    if i == len(pattern):
        return j - len(pattern)
    return -1
```

目标串无需回退，将失配位置继续与P[next[i]]位比较

返回模式串在T中的起始位置

边界情况：i == -1 代表了什么？

- 即在上一轮匹配中，i = 0，next[0] = -1， 而后 i=next[i] 赋值为-1
- 表示模式串中第一个字符的比较就发生失配
- T也无需再比较当前位置，向后移动一位
- 将next[0]设置为-1，让程序多循环一轮，就方便地实现了这一点



KMP匹配过程的时间复杂度

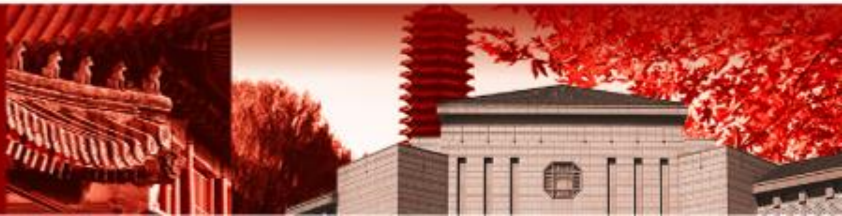
```python
def match(target, pattern, Next):
    i, j = 0, 0
    while i < len(pattern) and j < len(target):
        if pattern[i] == target[j] or i == -1:
            i += 1
            j += 1
        else:
            i = Next[i]
    if i == len(pattern):
        return j - len(pattern)
    return -1
```

并非每次循环 i 和 j 都会增加，无法通过循环次数直接判断

j 始终是不减的，最多执行O(n)次，n为target长度。满足if 条件的代码块执行次数也为O(n)

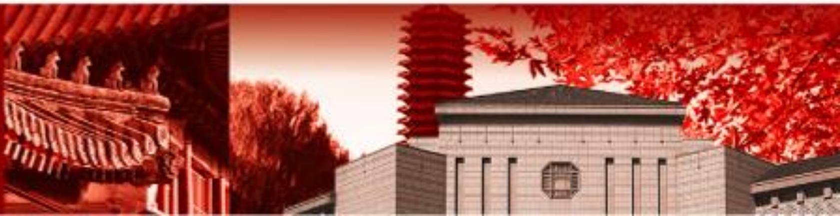
这部分执行了多少次？运用相同的推理：i = Next[i] 的次数不超过 i += 1的次数，即也为O(n)次

总结：时间复杂度为O(n)，n为target长度



KMP算法的时间复杂性分析

- 总结：
  - –假设pattern的长度为m，target的长度为n（通常m<n）
  - –计算Next数组的时间复杂度为O(m)
  - –匹配过程的时间复杂度为O(n)
  - –KMP算法总体的时间复杂度为O(m+n)
    - 对比：朴素模式匹配算法的时间复杂度为O(m*n)，这是一个显著的改进！



KMP算法：改进Next数组

- 考虑P[i]与T[j]的失配：
  - 此时要将P右移(i-k)位，继续比较T[j]与P[k]（k=Next[i]）
- 如果P[i] = P[k]？
  - 也是必定失配的，因而要继续右移，
  - 继续比较T[j]与P[k’]（k’=Next[k]）
  - 提前发现并优化这一过程

T[j]

✖

P[0]	**……**	**P[k-1]**	**P[k]**	**……**	**P[i-k-1]**	**P[i-k]**	**……**	**P[i-1]**	**P[i]**

P[0]	**……**	**P[k-1]**	**P[k]**

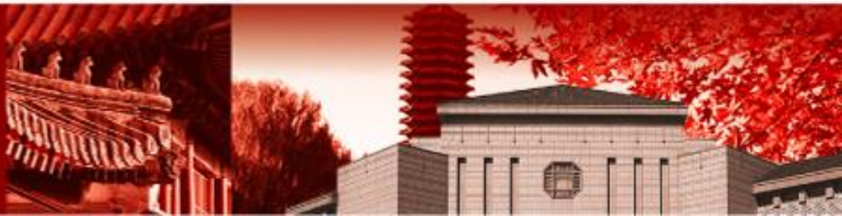
Next数组的计算—算法改进

```python
def improved_make_next(pattern):
    Next = [None] * len(pattern)
    i, k = 0, -1
    while i < len(pattern) - 1:
        if i == 0:
            Next[i] = -1
        while k >= 0 and pattern[i] != pattern[k]:
            k = Next[k]
        k += 1
        i += 1

        if p[i] == p[k]:
            Next[i] = Next[k]
        else:
            Next[i] = k
    return Next
```

优化：如果P[i]与P[k]相同，就优化Next[i]的值

注意，Next[k]也可能需要优化，但是一定在计算Next[i]之前就完成优化了



KMP算法示例：Next数组含义

数组下标k	0	1	2	3	4	5	6	7	8
模式串p	**a**	**b**	**c**	**a**	**a**	**b**	**a**	**b**	**c**
最大的相同前后缀长度	**0**	**0**	**0**	**1**	**1**	**2**	**1**	**2**	**3**
Next[k]	**-1**	**0**	**0**	**0**	**1**	**1**	**2**	**1**	**2**

右移一位，左边填-1

KMP算法示例：Next数组计算

假设已经计算得到Next[i]，如何计算Next[i+1]?

情况1： if **p[i] == p[k]**

Next[i+1] = k + 1;

数组下标 k	0	1	2	3	4	5	6	7	8
		k=1							
模式串p	**a**	**b**	**c**	**a**	**a**	**b**	**a**	**b**	**c**
Next[k]	**-1**	**0**	**0**	**0**	**1**	**1**	**2**		
						i=5			

KMP算法示例：Next数组计算

假设已经计算得到Next[i]，如何计算Next[i+1]?

情况2:   if **p[i] != p[k]**

k = Next[k]

数组下标 k	0 k=0	1	2 k=2	3	4	5	6	7	8
模式串p	**a**	**b**	**c**	**a**	**a**	**b**	**a**	**b**	**c**
Next[k]	**-1**	**0**	**0**	**0**	**1**	**1**	**2** i=6	**1**	

KMP算法示例

如何利用Next数组快速进行字符串匹配?

基本匹配思想:

情况1：主串字符=模式串字符，两边位置++

情况2：主串字符!=模式串字符，主串位置不变，模式串指向Next[k]

情况2特例：如果Next[k]=-1，那么主串位置++，模式串指向0

主串当前匹配位置指针不须回退，效率高

数组下标k	0	1	2
模式串p	**a**	**a**	**b**
Next[k]	**-1**	**0**	**1**

主串t	**a**	**b**	**a**	**a**	**a**	**a**	**b**	**a**
模式串p	**a**	**a**	**b**					

KMP算法示例

改进后的Next数组的匹配过程

基本匹配思想：

情况1：主串字符=模式串字符，两边位置++

情况2：主串字符!=模式串字符，主串位置不变，模式串指向Next[k]

情况2特例：如果Next[k]=-1，那么主串位置++，模式串指向0

改进版模式串的移位幅度更大，匹配效率更高

数组下标k	0	1	2
模式串p	a	a	b
Next[k]	-1	-1	1

主串t	a	b	a	a	a	a	b	a
模式串p	a	a	b					

Any Questions?