



# 数据结构和算法

## (Python描述)

郭 炜

微信公众号



微博: <http://weibo.com/guoweiofpku>

**学会程序和算法，走遍天下都不怕!**

讲义照片均为郭炜拍摄



北京大学  
PEKING UNIVERSITY

信息科学技术学院

北京大学信息学院 郭炜

# KMP 字符串匹配算法



法国勃朗峰

# 字符串匹配

给定一个模式串(子串) 和一个母串, 求模式串在母串中出现的位置。母串中找不到模式串位置就算-1。

# 暴力字符串匹配算法

引入"母串匹配起点" (MS) 概念。母串 $a$ , 子串 $b$ , 若 $a[i]$ 和 $b[0]$ 比较, 则称"母串匹配起点"为 $a[i]$ 。

设置一个母串指针, 指向母串中待比较的字符; 设置一个模式串指针, 指向子串中待比较的字符。比较母串指针和子串指针指向的字符, 如果相等, 则两个指针都加1。

如果不相等.....

# 暴力字符串匹配算法

暴力算法母串指针需要回溯。假设子串前 $n-1$ 个字符已经被匹配，第 $n$ 个失配：

MS  
母串： $a_0 a_1 a_2 \dots a_{n-1} a_n \dots$   
子串： $b_0 b_1 b_2 \dots b_{n-1} b_n \dots$

母串匹配起点+1，母串指针回溯到母串匹配起点：

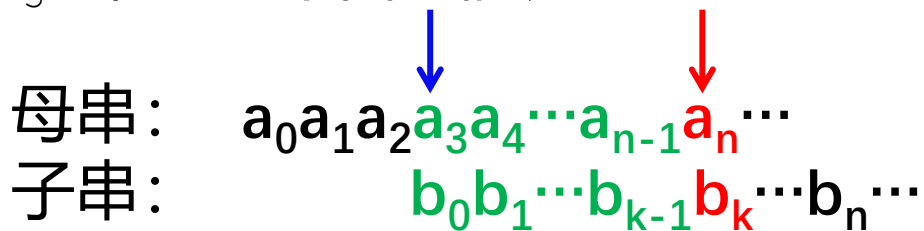
MS  
母串： $a_0 a_1 a_2 \dots a_{n-1} a_n \dots$   
子串： $b_0 b_1 \dots b_{n-2} b_{n-1} \dots$

# KMP字符串匹配算法

不希望母串指针回溯!

母串指针如果不回溯, 直接用 $a_n$ 和 $b_0$ 比, 可能就会忽略母串匹配起点为 $a_3$ 时得到的下面的情况:

母串:  $a_0a_1a_2$   $a_3a_4\cdots a_{n-1}$   $a_n\cdots$   
子串:  $b_0b_1\cdots b_{k-1}$   $b_k\cdots b_n\cdots$



$a_3a_4\cdots a_{n-1}$  和  $b_0b_1\cdots b_{k-1}$  可以匹配上, 值得再做下去

**关键:** 要在母串指针不回溯的情况下避免忽略上述情况。

# KMP字符串匹配算法

## ➤ 概念：字符串的前缀和后缀

$b_0b_1 \dots b_{n-1}b_n$  的前缀是  $b_0b_1 \dots b_k$  ( $k=0, 1, 2 \dots n$ )


$b_0b_1 \dots b_{n-1}b_n$  的真前缀是  $b_0b_1 \dots b_k$  ( $k=0, 1, 2 \dots n-1$ )

$b_0b_1 \dots b_{n-1}b_n$  的后缀是  $b_k \dots b_{n-1}b_n$  ( $k=0, 1, 2 \dots n$ )

$b_0b_1 \dots b_{n-1}b_n$  的真后缀是  $b_k \dots b_{n-1}b_n$  ( $k=1, 2 \dots n$ )

# KMP字符串匹配算法

若发生了下述情况：


  
 母串：  $a_0a_1a_2a_3a_4\cdots a_{n-1}a_n\cdots$ 
  
 子串：  $b_0b_1\cdots b_{k-1}b_kb_{k+1}\cdots b_n\cdots$ 
  
 $a_3a_4\cdots a_{n-1}$  和  $b_0b_1\cdots b_{k-1}$  可以匹配上

则  $b_0b_1\cdots b_{k-1}$  是子串的前缀，也是  $a_0a_1\cdots a_{n-1}$  的后缀

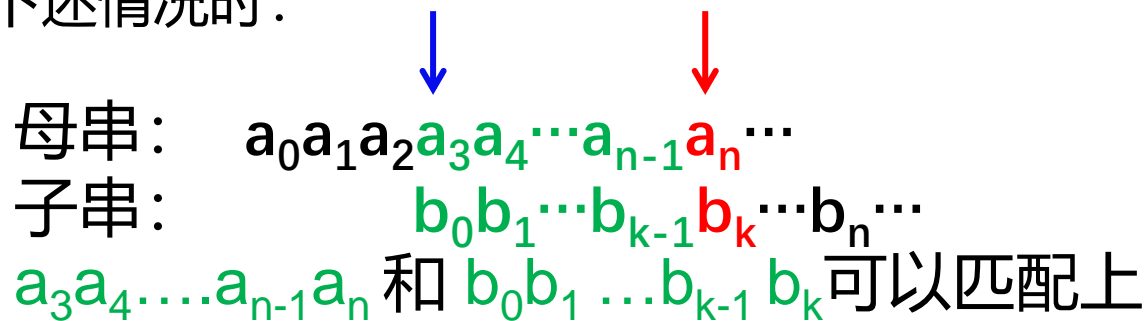
由于  $a_0a_1\cdots a_{n-1} = b_0b_1\cdots b_{n-1}$ ，因此  $b_0b_1\cdots b_{k-1}$  也是  $b_0b_1\cdots b_{n-1}$  的后缀

若有  $b_0b_1\cdots b_{k-1}$  是  $b_0b_1\cdots b_{n-1}$  的真后缀，则称  $b_0b_1\cdots b_{k-1}$  是字符  $b_n$  的前后缀。



# KMP字符串匹配算法

发生了下述情况时：



 母串：  $a_0 a_1 a_2 a_3 a_4 \cdots a_{n-1} a_n \cdots$   
 子串：  $b_0 b_1 \cdots b_{k-1} b_k \cdots b_n \cdots$   
 $a_3 a_4 \cdots a_{n-1} a_n$  和  $b_0 b_1 \cdots b_{k-1} b_k$  可以匹配上

字符 $b_n$ 的前后缀可能不止一个，要考查最长的

即 $a_n$ 和 $b_n$ 失配时，直接比较 $a_n$ 和 $b_n$ 最长前后缀的后一个字符 $b_k$ ，即母串指针不回溯，子串指针移到 $k$ 。若还失配，则比较 $a_n$ 和 $b_k$ 的最长前后缀（即 $b_n$ 的次长前后缀）后一个字符 $b_p$ ；若再失配，则比较 $a_n$ 和 $b_p$ 最长前后缀（即 $b_n$ 的第三长前后缀）的后一个字符 $\dots\dots$ 直到 $a_n$ 和 $b_0$ 比较，若还失配，则母串指针+1，子串指针回0。

# KMP字符串匹配算法

发生了下述情况时：


  
 母串：  $a_0a_1a_2a_3a_4\cdots a_{n-1}a_n\cdots$   
 子串：  $b_0b_1\cdots b_{k-1}b_k\cdots b_n\cdots$   
 $a_3a_4\cdots a_{n-1}a_n$  和  $b_0b_1\cdots b_{k-1}b_k$  可以匹配上

母串匹配位置从 $a_0$ 跳到了 $a_3$ ，忽略了母串匹配位置在 $a_1$ 和 $a_2$ 的情况，即忽略了母串分别从 $a_1$ 和 $a_2$ 开始与子串进行比较的情况。

如果能证明母串分别从 $a_1$ 和 $a_2$ 开始与子串进行比较，都不会使得 $a_1\cdots a_{n-1}$ 被匹配成功，则忽略是合理的。

# KMP字符串匹配算法

反证法：

母串：  $a_0 a_1 a_2 a_3 a_4 \dots a_{n-1} a_n \dots$   
 子串：  $b_0 b_1 b_2 b_3 \dots b_{m-1} b_m \dots$   
 $a_3 a_4 \dots a_{n-1} a_n$  和  $b_0 b_1 \dots b_{k-1} b_k$  可以匹配上

假设母串分别从  $a_1$  开始与子串进行比较,  $a_1 \dots a_{n-1}$  和  $b_0 \dots b_{m-1}$  匹配成功, 则  $b_0 b_1 b_2 b_3 \dots b_{m-1}$  是  $a_0 a_1 a_2 a_3 a_4 \dots a_{n-1}$  的后缀, 即是也是子串  $b_0 b_1 b_2 b_3 \dots b_{n-1}$  的后缀 (因  $a_0 a_1 a_2 a_3 a_4 \dots a_{n-1} = b_0 b_1 b_2 b_3 \dots b_{n-1}$ ), 即为字符  $b_n$  的前后缀。  $\text{len}(b_0 b_1 b_2 b_3 \dots b_{m-1}) > \text{len}(b_0 b_1 \dots b_{k-1})$ , 这和  $b_0 b_1 \dots b_{k-1}$  是字符  $b_n$  的最长前后缀矛盾。

因此母串匹配位置为  $a_1$  时不可能成功。

# KMP字符串匹配算法

用 $\text{next}[i]$ 表示字符 $b_i$ 的最长前后缀的长度, 则字符 $b_i$ 的次长前后缀的长度为 $\text{next}[\text{next}[i]]$ , 第三长前后缀的长度为 $\text{next}[\text{next}[\text{next}[i]]] \dots$

设 $b_i$ 最长前后缀长度为 $k$ , 则 $b_0b_1 \dots b_{k-1}$ 是最长前后缀, 和

$b_{i-k} \dots b_{i-1}$ 相同。  $b_i$ 次长前后缀就是 $b_k$ 的最长前后缀

故 $\text{Len}(b_i \text{次长前后缀}) = \text{next}[k] = \text{next}[\text{next}[i]]$

# KMP字符串匹配算法

## ➤ 算法核心思想：

设立列表`next`。`next[i]`就是`b[i]`的最长前后缀的长度。

`next[i]`表示匹配到子串字符`b[i]`时，若发生了失配，则母串指针不动，子串指针应该变为`next[i]`，然后继续匹配，再失配，则子串指针变为`next[next[i]]`.....

显然，`next`只和子串有关，和母串无关，且有  $\text{next}[1] = 0$   
记 $\text{next}[0] = -1$  (哨兵)，则 $\text{next}[i] \geq 0 \quad (i = 1, 2, \dots)$

若即`b[0]`失配，则母串指针+1，子串指针置为0

# KMP字符串匹配算法


KMP算法是如何避免母串指针回溯的？

母串：     acabaca<sup>↓</sup>kg.....  
子串：     acabacaef...

# KMP字符串匹配算法

KMP算法是如何避免母串指针回溯的？


母串： acabaca**k**g.....  
子串：        acab**a**caef...



# KMP字符串匹配算法

KMP算法是如何避免母串指针回溯的？

母串： acabaca**k**g.....  
子串：       a**c**abacaef...

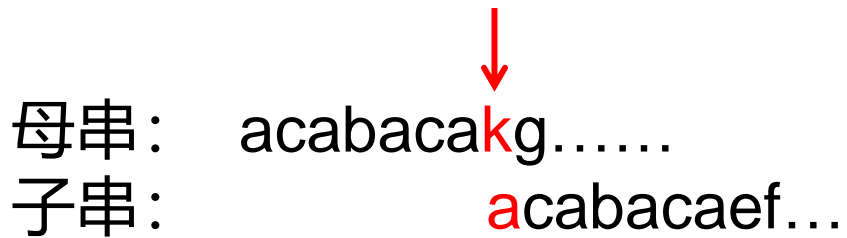




# KMP字符串匹配算法

KMP算法是如何避免母串指针回溯的？


母串： acabaca**k**g.....  
子串：           **a**cabacaef...



# KMP字符串匹配算法

KMP算法是如何避免母串指针回溯的？

母串： aabcdaakg.....  
子串：           acabacaef...



# 求next列表

字符串b: acabacaef

next: [-1, 0, 0, 1, 0, 1, 2, 3, 0]

next[i]: 字符b[i]的最长前后缀长度

# 求next列表

```
def kmp(a,b,Next):#a是母串, b是子串
    La,Lb = len(a),len(b)
    pa = pb = 0 #母串指针和子串指针
    while pa < La and pb < Lb:
        if pb == -1 or a[pa] == b[pb]:
            #pb==-1说明b[0]失配,因为只有next[0]才为-1
            pa,pb = pa + 1,pb + 1
        else:
            pb = Next[pb] #执行次数不会多于ps+1的执行次数, 即pa+1次数
    if pb == Lb:
        return pa - pb
    return -1
```

在Next列表已经算出的情况下, kmp复杂度 $O(La)$

# 求next列表

➤ 关键：求子串b的next列表

递推：已知 $\text{next}[0], \text{next}[1] \dots \text{next}[i]$ ，如何求 $\text{next}[i+1]$

设 $\text{next}[i] = k$ ，则若  $b[i] = b[k]$  则  $\text{next}[i+1] = k + 1$

$$b_0 b_1 b_2 \dots b_{k-1} b_k \dots b_{i-1} b_i b_{i+1}$$

若 $\text{next}[i] = k$ ，说明 $b_0 b_1 b_2 \dots b_{k-1} = b_{i-k} \dots b_{i-1}$

此时若 $b[i] = b[k]$ ，则 $b_0 b_1 b_2 \dots b_{k-1} b_k = b_{i-k} \dots b_{i-1} b_i$

即 $\text{next}[i+1] = k+1$

# 求next列表

设 $\text{next}[i] = k$ , 则若  $b[i] \neq b[k]$  则?

$b[i+1]$ 的最长前后缀, 必然是以 $b[i]$ 的某个前后缀加上 $b[i]$ 构成

如果 $b[i]$ 的最长前后缀(长度 $k$ ), 加上 $b[i]$ , 不能构成 $b[i+1]$ 的最长前后缀

则要考虑 $b[i]$ 的次长前后缀(长度为 $\text{next}[k]$ ), 能否加上 $b[i]$ , 构成 $b[i+1]$ 的最长前后缀, 次长的不行, 则考虑次次长前后缀(长度为 $\text{next}[\text{next}[k]]$ ).....

最终看 $b[0]$ 能否成为 $b[i+1]$ 最长前后缀

# 求next列表

设  $\text{next}[i] = k$ , 则若  $b[i] \neq b[k]$  则?

```
def countNext(b):
    i, k, Lb = 0, -1, len(b)
    Next = [-1 for i in range(Lb)]
    while i < Lb - 1:
        if k == -1 or b[i] == b[k]:
            Next[i+1] = k + 1
            i, k = i + 1, k + 1
        else:
            k = Next[k]
    return Next
```

复杂度  $O(\text{len}(b))$

# 求next列表(改进)

```
def countNext(b):  
    i,k,Lb = 0,-1,len(b)  
    Next = [-1 for i in range(Lb)]  
    while i < Lb - 1:  
        if k == -1 or b[i] == b[k]:  
            Next[i+1] = k + 1  
            i,k = i + 1,k + 1  
        else:  
            k = Next[k]  
    return Next
```

考虑 $b[k+1] == b[i+1]$ 成立的情况：若母串字符 $c$ 和 $b[i+1]$ 比较失配，也必然会和 $b[k+1]$ 比较失配，和 $b[k+1]$ 比较失配后，子串指针必然要回溯到 $Next[k+1]$ 。

复杂度 $O(\text{len}(b))$



# 求next列表(改进)

```
def countNext(b):  
    i,k,Lb = 0,-1,len(b)  
    Next = [-1 for i in range(Lb)]  
    while i < Lb - 1:  
        if k == -1 or b[i] == b[k]:  
            Next[i+1] = k + 1  
            i,k = i + 1,k + 1  
        else:  
            k = Next[k]  
    return Next
```

既然如此，当初就没有必要将子串指针回溯到k+1让c和b[k+1]去做比较，应该在c和b[i+1]失配时，直接将子串指针回溯到Next[k+1]

复杂度 $O(\text{len}(b))$

# 求next列表(改进)

```
def countNext(b):  
    i, k, L = 0, -1, len(b) # 开始k就是Next[0]  
    Next = [-1] * L  
    while i < L - 1:  
        if k == -1 or b[i] == b[k]:  
            if b[i+1] == b[k+1]:  
                Next[i+1] = Next[k+1]  
            else:  
                Next[i+1] = k+1  
            i, k = i + 1, k + 1  
        else:  
            k = Next[k]  
    return Next  
print(countNext("abbcabcaabbcaa"))  
#>> [-1, 0, 0, 0, -1, 0, 2, -1, 1, 0, 0, 0, -1, 5]
```

若 $b[k+1] == b[i+1]$ ，则将  
Next[i+1]设置为 Next[k+1]比  
设置为 k+1效率更高。

# 求next列表 (改进)

“abbcabcaabbcaa”

表 9.3.2 字符串“abbcabcaabbcaa”的 Next 列表

下标	0	1	2	3	4	5	6	7	8	9	10	11	12	13
字符串	a	b	b	c	a	b	c	a	a	b	b	c	a	a
Next	-1	0	0	0	0	1	2	0	1	1	2	3	4	5
Next2	-1	0	0	0	-1	0	2	-1	1	0	0	0	-1	5

Next[4]为0,但 $s[4] \neq s[0]$ , 所以母串字符若与 $s[4]$ 比较失配, 也没必要去和 $s[0]$ 比较, 应直接将子串指针回溯至Next[0], 即-1,故Next2[4]=-1。