



## Data Structure & Algorithm Analysis

# String

---

**Zibin Zheng ( 郑子彬 )**

**School of Data and Computer Science , SYSU**

**<http://www.inpluslab.com>**

课程主页: <http://inpluslab.sysu.edu.cn/dsa2016/>

---

线性表——具有相同类型的数据元素的有限序列。



限制插入、删除位置

{ 栈——仅在表的一端进行插入和删除操作  
  队列——在一端进行插入操作，而另一端进行删除操作

---

**串**——零个或多个字符组成的有限序列



将元素类型限制为字符

**线性表**——具有相同类型的数据元素的有限序列。



将元素类型扩充为线性表

(多维) **数组**——线性表中的数据元素可以是线性表

# String ( 字符串 )

---

- In computer programming, a string is traditionally a sequence of characters, either as a literal constant or as some kind of variable.
- A string is generally understood as a data type and is often implemented as an **array of bytes** (or words) that stores a sequence of elements, typically **characters**, using some **character encoding**.
- 在非数值处理、事务处理等问题常涉及到一系列的字符操作。计算机的硬件结构主要是反映**数值计算**的要求，因此，字符串的处理比具体数值处理复杂。本章讨论串的存储结构及几种基本的处理。

# Basic concepts of string

---

- 字符串(String)是零个或多个字符组成的有限序列。一般记作  $S = \text{“}a_1a_2a_3\dots a_n\text{”}$ , 其中  $S$  是串名, 双引号括起来的字符序列是串值;  $a_i (1 \leq i \leq n)$  可以是字母、数字或其它字符;
- 串中所包含的字符个数称为该串的长度。长度为零的串称为空串 (Empty String), 它不包含任何字符。
- 通常将仅由一个或多个空格组成的串称为空白串 (Blank String)
- 注意: 空串和空白串的不同, 例如“ ”和“”分别表示长度为1的空白串和长度为0的空串。

# Basic concepts of string

---

字符串的表示:

$$s = "a_1a_2...a_n" \quad (n \geq 0)$$

$$s1 = ""$$

$$s2 = " \quad "$$

- $s1$ 中没有字符，是一个空串；而  $s2$ 中有若干个空白格字符，它的长度不等于0，它是由空格字符组成的串，一般称此为空格串。

# Basic concepts of string

---

- 串中任意连续的字符组成的子序列被称为该串的子串。
- 包含子串的串又被称为该子串的主串。

a = "Welcome to Beijing"

b = "Welcome"

c = "Bei"

d = "Welcome to"

- 两个串相等：两个串的长度相等，并且各个对应的字符也都相同。
- 例如，有下列四个串a, b, c, d:

a = "program"

b = "Program"

c = "pro"

d = "program "

# Basic concepts of string

---

- 字符(char) : 组成字符串的基本单位 。
- 在C和C + + 中
  - 单字节 ( 8 bits )
  - 采用ASCII码对128个符号 ( 字符集charset ) 进行编码



## 串的逻辑结构

---

- 串的数据对象约束为某个字符集。
- 微机上常用的字符集是标准ASCII码，由 7 位二进制数表示一个字符，总共可以表示 128 个字符。
- 扩展ASCII码由 8 位二进制数表示一个字符，总共可以表示 256 个字符，足够表示英语和一些特殊符号，但无法满足国际需要。
- Unicode由 16 位二进制数表示一个字符，总共可以表示  $2^{16}$  个字符，能够表示世界上所有语言的所有字符，包括亚洲国家的表意字符。为了保持兼容性，Unicode字符集前256个字符与扩展ASCII码完全相同。

## Standard string in C++

---

- 串结束标记：'\0'
  - '\0'是ASCII码中8位BIT全0码，又称为**NULL**符。

# Standard string in C++

---

- 1. 串长函数

```
int strlen(char *s);
```

- 2. 串复制

```
char *strcpy(char *s1, char*s2);
```

- 3 . 串拼接

```
char *strcat(char *s1, char *s2);
```

- 4 . 串比较

```
int strcmp(char *s1, char *s2);
```

- 5 . 输入和输出函数

- 6 . 定位函数

```
char *strchr(char *s, char c);
```

- 7 . 右定位函数

```
char *strrchr(char *s, char c);
```

## Standard string in C++

---

- For example, a string **S**

“The quick brown dog jumps over the lazy fox ”

- Find the character 'r', `strchr(s, 'r');` return 11.
- Inversely find the character 'r', `strrchr(s, 'r');` return 29.

# Abstract data type of string

---

## ADT String

数据对象： $D = \{ a_i | a_i \in \text{CharacterSet}, i=1,2,\dots,n, n \geq 0 \}$

数据关系： $R = \{ \langle a_{i-1}, a_i \rangle | a_{i-1}, a_i \in D, i=2,3,\dots,n \}$

### • 字符串的基本操作

( 1 ) 创建串 StringAssign (s, string\_constant)

( 2 ) 判断串是否为空 StringEmpty(s)

( 3 ) 计算串长度 Length(s)

( 4 ) 串连接 Concat(s1, s2)

( 5 ) 求子串 SubStr(s1,s2start,len)

( 6 ) 串的定位 Index(s1,s2)

( 7 ) 字串的插入和删除

# Storage & implementation of string

---

串是一种特殊的线性表，其存储表示和线性表类似，但又不完全相同。串的存储方式取决于将要对串所进行的操作。串在计算机中有3种表示方式：

- ◆ **定长顺序存储表示**：将串定义成字符数组，利用串名可以直接访问串值。用这种表示方式，串的存储空间在编译时确定，其大小不能改变。

- ◆ **堆分配存储方式**：仍然用一组地址连续的存储单元来依次存储串中的字符序列，但串的存储空间是在程序运行时根据串的实际长度动态分配的。

- ◆ **块链存储方式**：是一种链式存储结构表示。

# Storage & implementation of string

---

- **串的定长顺序存储表示**
- 这种存储结构又称为**串的顺序存储结构**。是用一组连续的存储单元来存放串中的字符序列。所谓定长顺序存储结构，是直接使用定长的字符数组来定义，数组的上界预先确定。

定长顺序存储结构定义为：

```
#define MAX_STRLEN 256  
  
typedef struct  
{ char str[MAX_STRLEN];  
  int length;  
} StringType;
```

# Storage & implementation of string

---

## • 串的联结操作

Status StrConcat ( StringType s, StringType t)

/\* 将串t联结到串s之后, 结果仍然保存在s中 \*/

{ int i, j;

if ((s.length+t.length)>MAX\_STRLEN)

Return ERROR; /\* 联结后长度超出范围 \*/

for (i=0 ; i<t.length ; i++)

s.str[s.length+i]=t.str[i]; /\* 串t联结到串s之后 \*/

s.length=s.length+t.length; /\* 修改联结后的串长度 \*/

return OK;

}



# Storage & implementation of string

---

## • 求子串操作

Status SubString (StringType s, int pos, int len, StringType \*sub)

```
{ int k, j ;  
    if (pos<1||pos>s.length||len<0||len>(s.length-pos+1))  
        return ERROR ; /* 参数非法 */  
    sub->length=len;  
    for (j=0, k=pos ; j<=len ; k++ , j++)  
        sub->str[j]=s.str[k] ; /* 逐个字符复制求得子串 */  
    return OK ;  
}
```

# Storage & implementation of string

---

## 串的堆分配存储表示

实现方法：系统提供一个空间足够大且地址连续的存储空间(称为“堆”)供串使用。可使用C语言的动态存储分配函数malloc()和free()来管理。

特点是：仍然以一组地址连续的存储空间来存储字符串值，但其所需的存储空间是在程序执行过程中动态分配，故是动态的，变长的。

串的堆式存储结构的类型定义

```
typedef struct
```

```
{   char *ch;   /* 若非空，按长度分配，否则为NULL */  
    int length; /* 串的长度 */  
} HString ;
```

# Storage & implementation of string

---

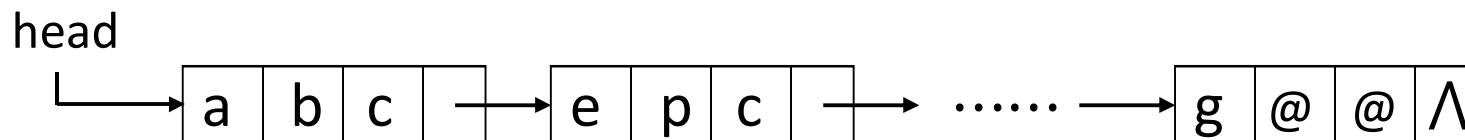
## • 串的联结操作

```
Status Hstring *StrConcat(HString *T, HString *s1, HString *s2)
/* 用T返回由s1和s2联结而成的串 */
{ int k, j, t_len;
  if (T.ch) free(T); /* 释放旧空间 */
  t_len=s1->length+s2->length;
  if ((p=(char *)malloc(sizeof(char)*t_len))==NULL)
    { printf( "系统空间不够, 申请空间失败 !\n" );
      return ERROR ; }
  for (j=0 ;j<s1->length;j++)
    T->ch[j]=s1->ch[j]; /* 将串s复制到串T中 */
  for (k=s1->length, j=0 ;j<s2->length; k++, j++)
    T->ch[k]=s1->ch[j]; /* 将串s2复制到串T中 */
  free(s1->ch);
  free(s2->ch);
  return OK ;
}
```

# Storage & implementation of string

## 链式存储表示

- 串的链式存储结构和线性表的串的链式存储结构类似，采用单链表来存储串，结点的构成是：
  - ◆ data域：存放字符，data域可存放的字符个数称为结点的大小；
  - ◆ next域：存放指向下一结点的指针。
- 若每个结点仅存放一个字符，则结点的指针域就非常多，造成系统空间浪费，为节省存储空间，考虑串结构的特殊性，使每个结点存放若干个字符，这种结构称为块链结构。如下图是块大小为3的串的块链式存储结构示意图。



串的块链式存储结构示意图

# Storage & implementation of string

---

串的块链式存储的类型定义包括：

(1) 块结点的类型定义

```
#define BLOCK_SIZE 4
typedef struct Blstrtype
{ char data[BLOCK_SIZE];
  struct Blstrtype *next;
}BNODE ;
```

(2) 块链串的类型定义

```
typedef struct
{ BNODE head; /* 头指针 */
  int Strlen; /* 当前长度 */
} Blstring ;
```

在这种存储结构下，结点的分配总是完整的结点为单位，因此，为使一个串能存放在整数个结点中，在串的末尾填上不属于串值的特殊字符，以表示串的终结。

当一个块(结点)内存放多个字符时，往往会使操作过程变得较为复杂，如在串中插入或删除字符操作时通常需要在块间移动字符。

## Pattern matching of string

---

- 子串在主串中的定位称为模式匹配(Pattern Matching)或串匹配(String Matching)。模式匹配成功是指在主串S中能够找到模式串T，否则，称模式串T在主串S中不存在。
- 此运算的应用在非常广泛。例如，在文本编辑程序中，我们经常要查找某一特定单词在文本中出现的位置。显然，解此问题的有效算法能极大地提高文本编辑程序的响应性能。
- 模式匹配是一个较为复杂的串操作过程。迄今为止，人们对串的模式匹配提出了许多思想和效率各不相同的计算机算法。

## Naïve pattern matching ( exhaustion )

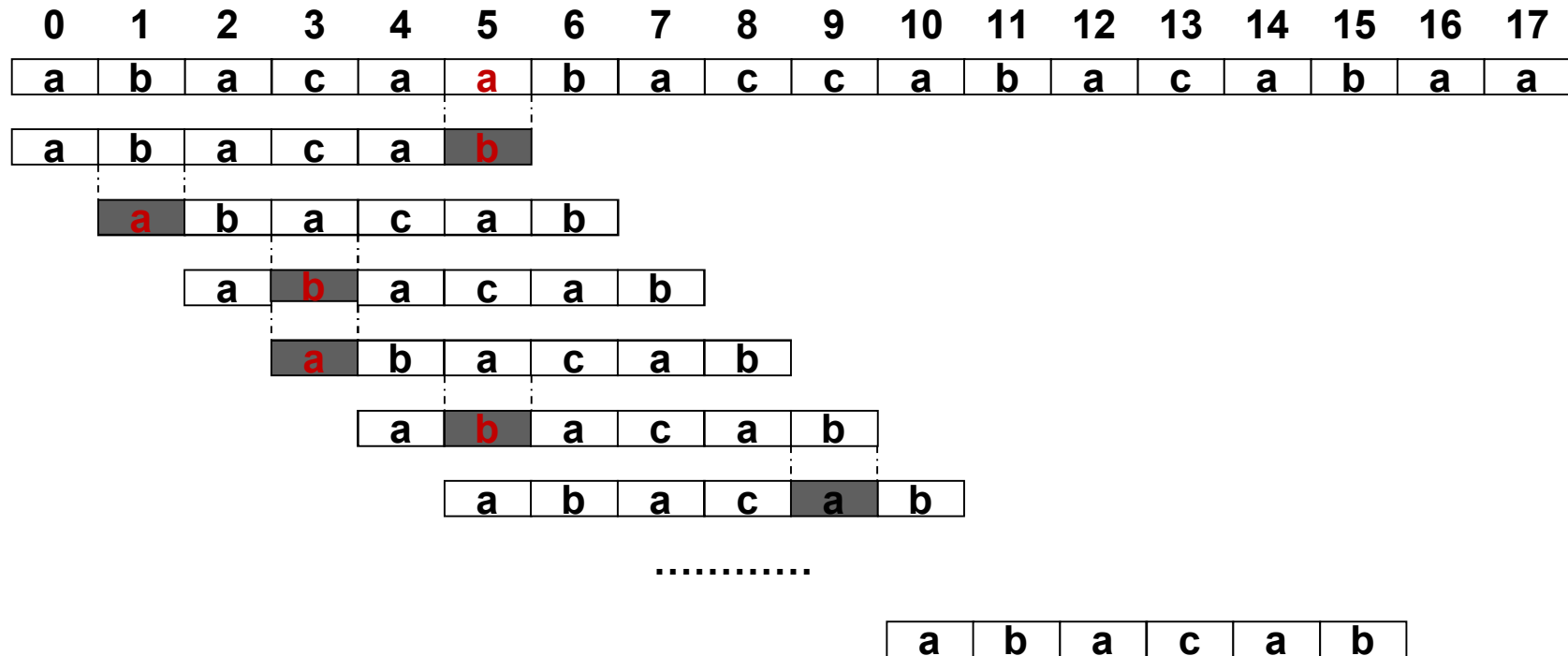
---

- 在串匹配中，一般将主串称为目标串，子串称之为模式串。  
设 $S$ 为目标串， $T$ 为模式串，且不妨设：

$$S = "s_1 s_2 \cdots s_{n-1}" \quad T = "t_1 \cdots t_{m-1}"$$

- 串的匹配实际上是对合法的位置  $1 \leq i \leq n-m$  依次将目标串中的子串  $S[i..i+m-1]$  和模式串  $T[1..m-1]$  进行比较
- 若  $S[i..i+m-1] = T[1..m-1]$ ，则称从位置  $i$  开始的匹配成功，亦称模式  $T$  在目标  $S$  中出现；
- 若  $S[i..i+m-1] \neq T[1..m-1]$ ，则称从位置  $i$  开始的匹配失败。

# Naïve pattern matching ( exhaustion )



- 把模式与目标逐一进行比较（首位置开始），直到碰到不匹配的字符为止（模式右移一位再次开始匹配）
- 算法可在第一个匹配或是目标的结束处停止



# Implementation

---

```
#include "String.h"
#include <assert.h>
int NaiveStrMatching (String T, String P) {
    int i = 0;                // 模式的下标变量
    int j = 0;                // 目标的下标变量
    int pLen = P.length( );   // 模式的长度
    int tLen = T.length( );   // 目标的长度
    if (tLen < pLen)           // 如果目标比模式短，匹配无法成功
        return (-1);
    while ( i < pLen && j < tLen) // 反复比较对应字符来开始匹配
        if (T[j] == P[i])
            i++, j++;
        else {
            j = j - i + 1;
            i = 0;
        }
    if ( i >= pLen)
        return (j - pLen + 1);
    else return (-1);
}
```

## Naïve pattern matching ( exhaustion ) : efficiency analysis

---

### 理解该算法的关键点

当第一次 $s_k \neq t_j$ 时：主串要退回到 $k-j+1$ 的位置，而模式串也要退回到第一个字符（即 $j=0$ 的位置）。

比较出现 $s_k \neq t_j$ 时：则应该有 $s_{k-1} = t_{j-1}$ ， $\dots$ ， $s_{k-j+1} = t_1$ ， $s_{k-j} = t_0$ 。

## Naïve pattern matching ( exhaustion ) : efficiency analysis

---

- 假定目标T的长度为n，模式P长度为m，且  $m \leq n$ 
  - 在最坏的情况下，每一次循环都不成功，则一共要进行比较  $(n-m+1)$  次
  - 每一次“相同匹配”比较所耗费的时间，是P和T逐个字符比较的时间，最坏情况下，共m次
  - 因此，整个算法的最坏时间开销估计为  $O(mn)$

# Naïve pattern matching: worst case

- 模式与目标的每一个长度为m的子串进行比较

AAAAA AAAAAAAAAAAAAAAAAA  
AAAAB 5次比较

AAAAA AAAAAAAAAAAAAAAAAA  
AAAAB 5次比较

AAAAA AAAAAAAAAAAAAAAAAA  
AAAAB 5次比较

AAAAA AAAAAAAAAAAAAAAAAA  
AAAAB 5次比较

.....

AAAAAAAAAAAAAAAAAAAAAAA  
AAAAA  
AAAAB 5次比较

- 目标形如 $a^n$ , 模式形如 $a^{m-1}b$
- 总比较次数:  $O(n-m+1)$
- 时间复杂度:  $O(mn)$

## Naïve pattern matching: best case--Matching

---

- 在目标的前 $m$ 个位置上找到模式，设  $m = 5$

AAAAAAAAAAAAAAAAAAAAAAAAAAAAB

AAAAA

5次比较

- 总比较次数:  $m$
- 时间复杂度:  $O(m)$

## Naïve pattern matching: best case—Not Matching

- 总是在第一个字符上不匹配

A A A A A A A A A A A A A A A A A A A H

O O O O H      1次比较

A A A A A A A A A A A A A A A A A A A H

O O O O H      1次比较

A A A A A A A A A A A A A A A A A A A H

O O O O H      1次比较

A A A A A A A A A A A A A A A A A A A H

O O O O H      1次比较

.....

A A A A A A A A A A A A A A A A A A A H

1次比较      O O O O H

- 总比较次数:  $n-m+1$

- 时间复杂度:  $O(n)$

## Case: naïve pattern matching

---

- 举个例子，如果给定文本串s “BBC ABCDAB ABCDABCDABDE”，和模式串P “ABCDABD”。

1. S[0]为B，P[0]为A，不匹配，执行第②条指令：“如果失配（即 $S[i] \neq P[j]$ ），令 $i = i - (j - 1)$ ， $j = 0$ ”，S[1]跟P[0]匹配，相当于模式串要往右移动一位（ $i=1$ ， $j=0$ ）

$i = i - j + 1$

i-j: 回到初始位置

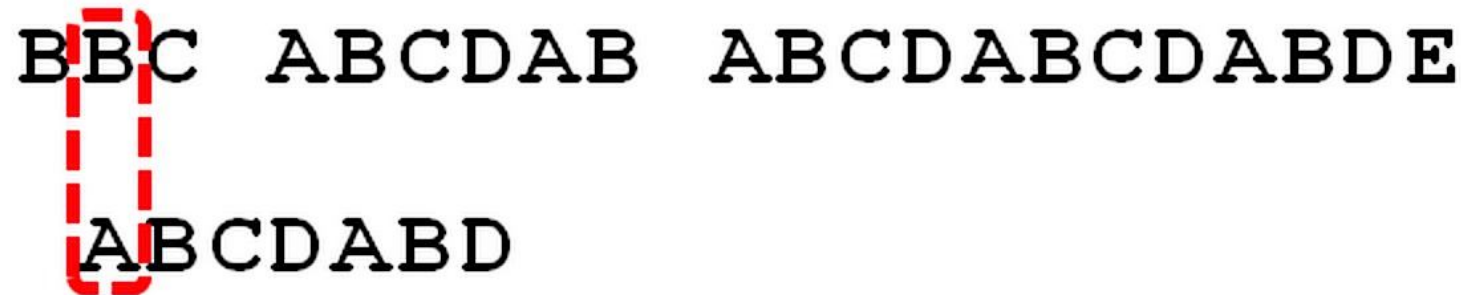
BBC ABCDAB ABCDABCDABDE  
ABCDABD

参考资料：从头到尾彻底理解KMP

[http://blog.csdn.net/qq\\_33583069/article/details/51922494](http://blog.csdn.net/qq_33583069/article/details/51922494)

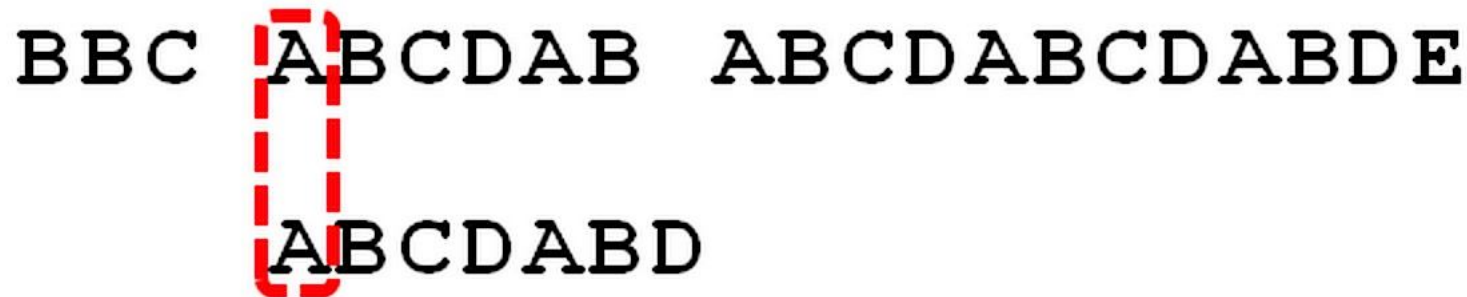
## Case: naïve pattern matching

2.  $S[1]$ 跟 $P[0]$ 还是不匹配，继续执行第②条指令：“如果失配（即 $S[i] \neq P[j]$ ），令 $i = i - (j - 1)$ ， $j = 0$ ”， $S[2]$ 跟 $P[0]$ 匹配（ $i=2$ ， $j=0$ ），从而模式串不断的向右移动一位（不断的执行“令 $i = i - (j - 1)$ ， $j = 0$ ”， $i$ 从2变到4， $j$ 一直为0）



BBC ABCDAB ABCDABCDABDE  
ABCDABD

3. 直到 $S[4]$ 跟 $P[0]$ 匹配成功（ $i=4$ ， $j=0$ ），此时按照上面的暴力匹配算法的思路，转而执行第①条指令：“如果当前字符匹配成功（即 $S[i] == P[j]$ ），则 $i++$ ， $j++$ ”，可得 $S[i]$ 为 $S[5]$ ， $P[j]$ 为 $P[1]$ ，即接下来 $S[5]$ 跟 $P[1]$ 匹配（ $i=5$ ， $j=1$ ）



BBC ABCDAB ABCDABCDABDE  
ABCDABD

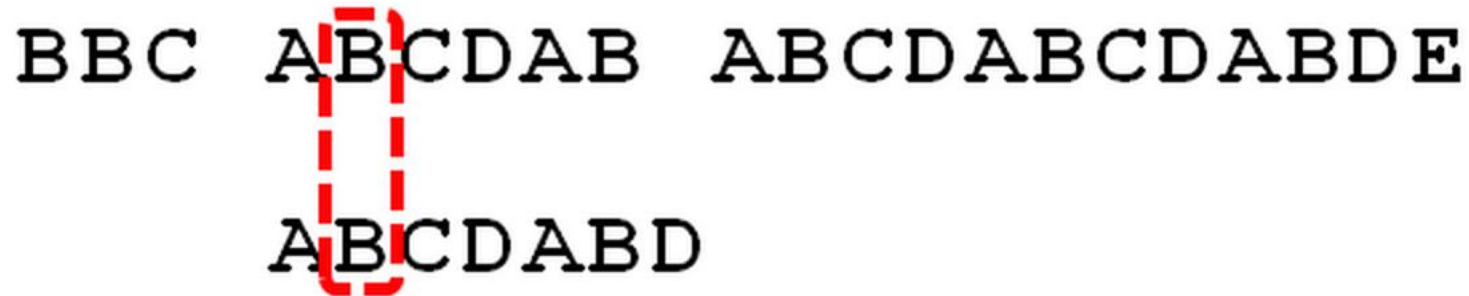


## Case: naïve pattern matching

---

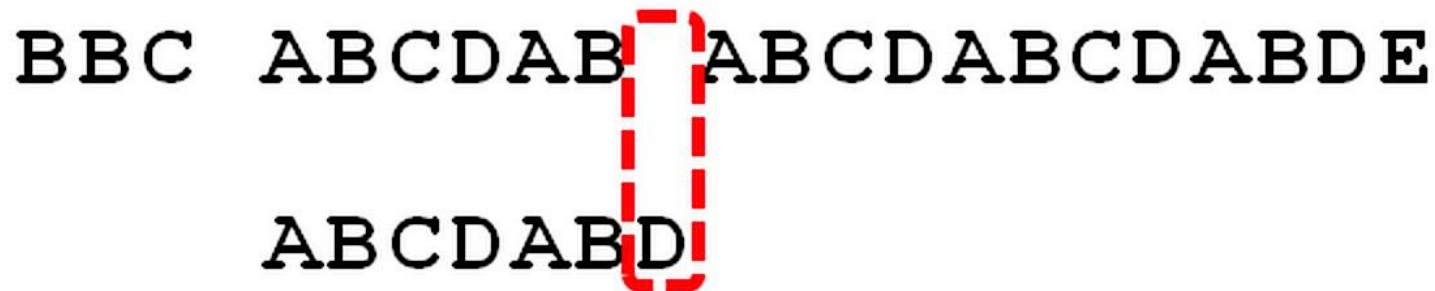
4. S[5]跟P[1]匹配成功，继续执行第①条指令：“如果当前字符匹配成功（即 $S[i] == P[j]$ ），则 $i++$ ， $j++$ ”，得到S[6]跟P[2]匹配（ $i=6$ ， $j=2$ ），如此进行下去

BBC ABCDAB ABCDABCDABDE  
ABCDABD



5. 直到S[10]为空格字符，P[6]为字符D（ $i=10$ ， $j=6$ ），因为不匹配，重新执行第②条指令：“如果失配（即 $S[i] \neq P[j]$ ），令 $i = i - (j - 1)$ ， $j = 0$ ”，相当于S[5]跟P[0]匹配（ $i=5$ ， $j=0$ ）

BBC ABCDAB ABCDABCDABDE  
ABCDABD



## Case: naïve pattern matching

---

6. 至此，我们可以看到，如果按照暴力匹配算法的思路，尽管之前文本串和模式串已经分别匹配到了S[9]、P[5]，但因为S[10]跟P[6]不匹配，所以文本串回溯到S[5]，模式串回溯到P[0]，从而让S[5]跟P[0]匹配。

BBC ABCDAB ABCDABCDABDE  
          ABCDABD

而S[5]肯定跟P[0]失配。为什么呢？因为在之前第4步匹配中，我们已经得知S[5] = P[1] = B，而P[0] = A，即P[1] != P[0]，故S[5]必定不等于P[0]，所以回溯过去必然会导致失配。那有没有一种算法，让i 不往回退，只需要移动j 即可呢？

能否尽可能用好之前的对比结果？

# An improved pattern matching--KMP algorithm

---

- **Knuth-Morris-Pratt 字符串查找算法**，简称为“KMP算法”，常用于在一个文本串S内查找一个模式串P 的出现位置，这个算法由Donald Knuth、Vaughan Pratt、James H. Morris三人同时独立发现，后取这3人的姓氏命名此算法。
- 每当一趟匹配过程出现字符不相等时，主串指示器不用回溯，而是利用已经得到的“部分匹配”结果，将模式串的指示器向右“**滑动**”尽可能远的一段距离后，继续进行比较。

# An improved pattern matching--KMP algorithm

继续拿之前的例子来说，当S[10]跟P[6]匹配失败时，KMP不是简单的如朴素匹配那样把模式串右移一位，而是执行第②条指令：“如果 $j \neq -1$ ，且当前字符匹配失败（即 $S[i] \neq P[j]$ ），则令 $i$ 不变， $j = \text{next}[j]$ ”，即 $j$ 从6变到2（后面我们将求得P[6]，即字符D对应的next 值为2），所以相当于模式串向右移动的位数为 $j - \text{next}[j]$ 位（ $j - \text{next}[j] = 6 - 2 = 4$ 位）。

BBC ABCDAB ABCDABCDABDE  
ABCDABD

向右移动4位后，S[10]跟P[2]继续匹配。为什么要向右移动4位呢，因为移动4位后，模式串中又有个“AB”可以继续跟S[8]S[9]匹配，相当于在模式串中找相同的前缀和后缀，然后根据前缀后缀求出next 数组，最后基于next 数组进行匹配（不关心next 数组怎么求来的，只想看匹配过程是咋样的，可直接跳到下文[3.3.4节](#)）。

BBC ABCDAB ABCDABCDABDE  
ABCDABD

# Steps of KMP algorithm

---

- ① 寻找前缀和后缀最长公共元素长度
- ② 求next数组
- ③ 匹配失配的处理

## • KMP的算法流程

假设现在文本串S匹配到  $i$  位置，模式串P匹配到  $j$  位置

- 如果  $j = -1$ ，或者当前字符匹配成功（即  $S[i] == P[j]$ ），都令  $i++$ ， $j++$ ，继续匹配下一个字符；
- 如果  $j \neq -1$ ，且当前字符匹配失败（即  $S[i] \neq P[j]$ ），则令  $i$  不变， $j = \text{next}[j]$ 。此举意味着失配时，模式串P相对于文本串S向右移动了  $j - \text{next}[j]$  位。
  - 换言之，当匹配失败时，模式串向右移动的位数为：失配字符所在位置 - 失配字符对应的next值（next数组的求解会在下文的3.3.3节中详细阐述），即移动的实际位数为： $j - \text{next}[j]$ ，且此值大于等于1。

# KMP algorithm

- 寻找前缀和后缀最长公共元素长度
  - 如果给定的模式串是：“ABCDABD”，从左至右遍历整个模式串，其各个子串的前缀后缀分别如下表格所示：

模式串的各个子串	前缀	后缀	最大公共元素长度
A	空	空	0
AB	A	B	0
ABC	A,AB	C,BC	0
ABCD	A,AB,ABC	D,CD,BCD	0
ABCD A	A,AB,ABC,ABCD	A,DA,CDA,BCDA	1
ABCDAB	A,AB,ABC,ABCD,ABCD A	B,AB,DAB,CDAB,BCDAB	2
ABCDABD	A,AB,ABC,ABCD,ABCD A ABCDAB	D,BD,ABD,DABD,CDABD BCDABD	0



# KMP algorithm

---

- 寻找前缀和后缀最长公共元素长度
  - 如果给定的模式串是：“ABCDABD”，则字符串对应的各个前缀后缀的公共元素的最大长度表为（下简称《最大长度表》）：

字符	A	B	C	D	A	B	D
最大前缀后缀公共元素长度	0	0	0	0	1	2	0

# KMP algorithm

---

- 基于《最大长度表》匹配

- 因为模式串中首尾可能会有重复的字符，故可得出下述结论：

失配时，模式串向右移动的位数为：

已匹配字符数 - 失配字符的上一位字符所对应的最大长度值

- 给定文本串 “BBC ABCDAB ABCDABCDABDE”，和模式串 “ABCDABD”

BBC ABCDAB ABCDABCDABDE  
ABCDABD



## KMP--基于《最大长度表》匹配

1. 因为模式串中的字符A跟文本串中的字符B、B、C、空格一开始就不匹配，所以不必考虑结论，直接将模式串不断的右移一位即可，直到模式串中的字符A跟文本串的第5个字符A匹配成功：

BBC ABCDAB ABCDABCDABDE  
ABCDABD

字符	A	B	C	D	A	B	D
最大前缀后缀公共元素长度	0	0	0	0	1	2	0

## KMP--基于《最大长度表》匹配

- 2. 继续往后匹配，当模式串最后一个字符D跟文本串匹配时失配，显而易见，模式串需要向右移动。但向右移动多少位呢？因为此时已经匹配的字符数为6个（ABCDAB），然后根据《最大长度表》可得失配字符D的上一位字符B对应的长度值为2，所以根据之前的结论，可知需要向右移动 $6 - 2 = 4$  位。

BBC ABCDAB ABCDABCDABDE  
          ABCDABD

字符	A	B	C	D	A	B	D
最大前缀后缀 公共元素长度	0	0	0	0	1	2	0

## KMP--基于《最大长度表》匹配

- 3. 模式串向右移动4位后，发现C处再度失配，因为此时已经匹配了2个字符（AB），且上一位字符B对应的最大长度值为0，所以向右移动： $2 - 0 = 2$  位。

BBC ABCDAB ABCDABCDABDE  
          ABCDABD

字符	A	B	C	D	A	B	D
最大前缀后缀 公共元素长度	0	0	0	0	1	2	0

## KMP--基于《最大长度表》匹配

- 4. A与空格失配，向右移动 1 位。

BBC ABCDAB ABCDABCDABDE  
                  ABCDABD

- 5. 继续比较，发现D与C 失配，故向右移动的位数为：已匹配的字符数6 减去上一位字符B对应的最大长度2，即向右移动  $6 - 2 = 4$  位。

BBC ABCDAB ABCDABCDABDE  
                          ABCDABD

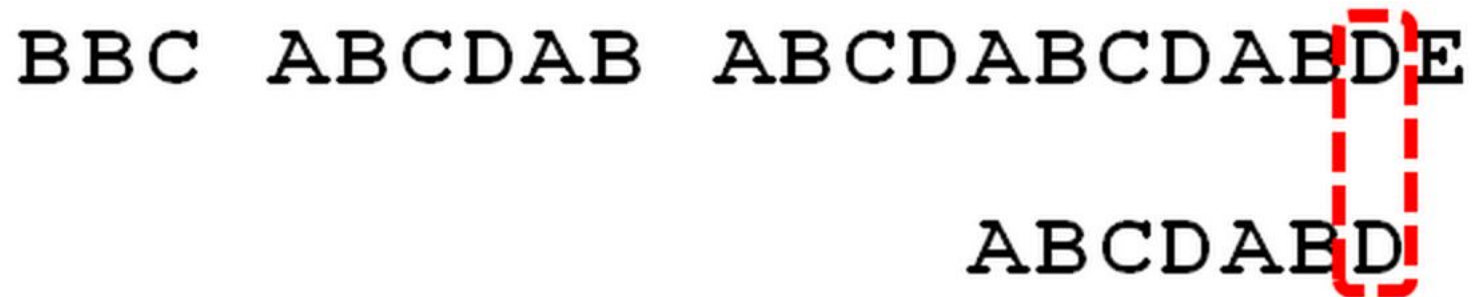
字符	A	B	C	D	A	B	D
最大前缀后缀 公共元素长度	0	0	0	0	1	2	0

## KMP--基于《最大长度表》匹配

---

- 6. 经历第5步后，发现匹配成功，过程结束。

BBC ABCDAB ABCDABCDABDE  
ABCDABD



## KMP--根据《最大长度表》求出next数组

- 已知字符串 “ABCDABD”各个前缀后缀的最大公共元素长度分别为：

模式串	A	B	C	D	A	B	D
前后缀最大公共元素长度	0	0	0	0	1	2	0

- 根据这个表可以得出下述结论：
  - 失配时，模式串向右移动的位数为：  
已匹配字符数 - 失配字符的上一位字符所对应的最大长度值

## KMP--根据《最大长度表》求出next数组

- 利用最大长度表和结论进行匹配时，可以发现，当匹配到一个字符失配时，其实没必要考虑当前失配的字符，每次失配时，都是使用失配字符的上一位字符对应的最大长度值。
- 如此，便引出了**next 数组**。
- 给定字符串 “ABCDABD”，可求得它的next 数组如下：

模式串	A	B	C	D	A	B	D
next	-1	0	0	0	0	1	2

## KMP--根据《最大长度表》求出next数组

- 把next 数组跟之前求得的最大长度表对比后，不难发现，next 数组相当于“最大长度值” 整体向右移动一位，然后初始值赋为-1。
- 换言之，对于给定的模式串：ABCDABD，它的最大长度表及next 数组分别如下：

模式串	A	B	C	D	A	B	D
最大长度值	0	0	0	0	1	2	0
next 数组	-1	0	0	0	0	1	2

- 根据最大长度表求出了next 数组后，从而有

失配时，模式串向右移动的位数为：失配字符所在位置 - 失配字符对应的next 值



# KMP--根据《next数组》匹配

## 1. 最开始匹配时

BBC ABCDAB ABCDABCDABDE  
ABCDABD

2. P[1]跟S[5]匹配成功，P[2]跟S[6]也匹配成功，...，直到当匹配到字符D时失配（即S[10] != P[6]），由于j从0开始计数，故数到失配的字符D时j为6，且字符D对应的next值为2，所以向右移动的位数为： $j - \text{next}[j] = 6 - 2 = 4$  位

BBC ABCDAB ABCDABCDABDE  
ABCDABD

字符	A	B	C	D	A	B	D
Next值	-1	0	0	0	0	1	2

## KMP--根据《next数组》匹配

3. 向右移动4位后，c再次失配，向右移动： $j - \text{next}[j] = 2 - 0 = 2$  位

BBC ABCDAB ABCDABCDABDE  
                  ABCDABD

4. 移动两位之后，A 跟空格不匹配，再次后移1 位

BBC ABCDAB ABCDABCDABDE  
                  ABCDABD

字符	A	B	C	D	A	B	D
Next值	-1	0	0	0	0	1	2

## KMP算法--根据《next数组》匹配

- 5. D处失配，向右移动  $j - \text{next}[j] = 6 - 2 = 4$  位

BBC ABCDAB ABCDABCDABDE  
ABCDABD

- 6. 匹配成功，过程结束。

BBC ABCDAB ABCDABCDABDE  
ABCDABD

字符	A	B	C	D	A	B	D
Next值	-1	0	0	0	0	1	2

# KMP算法实现

```
1. int KmpSearch(char* s, char* p)
2. {
3.     int i = 0;
4.     int j = 0;
5.     int sLen = strlen(s);
6.     int pLen = strlen(p);
7.     while (i < sLen && j < pLen)
8.     {
9.         //①如果j = -1, 或者当前字符匹配成功 (即S[i] == P[j]) , 都令i++, j++
10.        if (j == -1 || s[i] == p[j])
11.        {
12.            i++;
13.            j++;
14.        }
15.        else
16.        {
17.            //②如果j != -1, 且当前字符匹配失败 (即S[i] != P[j]) , 则令 i 不变, j = next[j]
18.            //next[j]即为j所对应的next值
19.            j = next[j];
20.        }
21.    }
22.    if (j == pLen)
23.        return i - j;
24.    else
25.        return -1;
26. }
```

- 
- 下面的问题是：已知 $\text{next}[0, \dots, j]$ ，如何求出 $\text{next}[j + 1]$ 呢？
  - 对于 $\text{pattern}$ 的前 $j+1$ 个序列字符：
  - 若 $\text{pattern}[k] == \text{pattern}[j]$ ，则 $\text{next}[j + 1] = \text{next}[j] + 1 = k + 1$ ；
  - 若 $\text{pattern}[k] \neq \text{pattern}[j]$ ，如果此时 $\text{pattern}[\text{next}[k]] == \text{pattern}[j]$ ，则 $\text{next}[j + 1] = \text{next}[k] + 1$ ，否则继续递归重复此过程。相当于在字符 $p[j+1]$ 之前不存在长度为 $k+1$ 的前缀“ $p_0 p_1, \dots, p_{k-1} p_k$ ”跟后缀“ $p_{j-k} p_{j-k+1}, \dots, p_{j-1} p_j$ ”相等，那么是否可能存在另一个值 $t+1 < k+1$ ，使得长度更小的前缀“ $p_0 p_1, \dots, p_{t-1} p_t$ ”等于长度更小的后缀“ $p_{j-t} p_{j-t+1}, \dots, p_{j-1} p_j$ ”呢？如果存在，那么这个 $t+1$ 便是 $\text{next}[j+1]$ 的值，此相当于利用 $\text{next}$ 数组进行P串前缀跟P串后缀的匹配。

- 如下图所示，假定给定模式串ABCDABCE，且已知 $\text{next}[j] = k$ （相当于“ $p_0 p_{k-1}$ ” = “ $p_{j-k} p_{j-1}$ ” = AB，可以看出k为2），现要求 $\text{next}[j+1]$ 等于多少？因为 $p_k = p_j = C$ ，所以 $\text{next}[j+1] = \text{next}[j] + 1 = k + 1$ （可以看出 $\text{next}[j+1] = 3$ ）。代表字符E前的模式串中，有长度k+1的相同前缀后缀。

模式串	A	B	C	D	A	B	C	E
前后缀 相同长 度	0	0	0	0	1	2	3	0
next 值	-1	0	0	0	0	1	2	?
索引	$p_0$	$p_{k-1}$	$p_k$	$p_{k+1}$	$p_{j-k}$	$p_{j-1}$	$p_j$	$p_{j+1}$

- 但如果  $p_k \neq p_j$  呢？说明“ $p_0 p_{k-1} p_k$ ”  $\neq$  “ $p_{j-k} p_{j-1} p_j$ ”。换言之，当  $p_k \neq p_j$  后，字符E前有多大长度的相同前缀后缀呢？很明显，因为C不同于D，所以ABC跟ABD不相同，即字符E前的模式串没有长度为k+1的相同前缀后缀，也就不能再简单的令： $next[j+1] = next[j] + 1$ 。所以，咱们只能去寻找长度更短一点的相同前缀后缀。

模式串	A	B	<u>C</u>	D	A	B	<u>D</u>	E
前后缀相同长度	0	0	0	0	1	2	0	0
next 值	-1	0	0	0	0	1	2	?
索引	$p_0$	$p_{k-1}$	$p_k$	$p_{k+1}$	$p_{j-k}$	$p_{j-1}$	$p_j$	$p_{j+1}$

- 
- 以换个角度思考这个问题：
  - 类似KMP的匹配思路，当 $p_0 p_1, \dots, p_j$ 跟主串 $s_0 s_1, \dots, s_i$ 匹配时，如果模式串在 $j$ 处失配，则 $j = \text{next}[j]$ ，相当于模式串需要向右移动 $j - \text{next}[j]$ 位。
  - 现在前缀“ $p_0 p_{k-1} p_k$ ”去跟后缀“ $p_{j-k} p_{j-1} p_j$ ”匹配，发现在 $p_k$ 处匹配失败，那么前缀需要向右移动多少位呢？根据已经求得的前缀各个字符的 $\text{next}$ 值，可得前缀应该向右移动 $k - \text{next}[k]$ 位，相当于 $k = \text{next}[k]$ 。
    - 若移动之后， $p_{k'} = p_j$ ，则代表字符 $E$ 前存在长度为 $\text{next}[k'] + 1$ 的相同前缀后缀；
  - 否则继续递归 $k = \text{next}[k]$ ，直到 $p_k$ 跟 $p_j$ 匹配成功，或者不存在任何 $k$ （ $0 < k < j$ ）满足 $p_k = p_j$ ，且 $k = \text{next}[k] = -1$ 停止递归。



# KMP算法实现--求得next数组

---

```
1. void GetNext(char* p,int next[])
2. {
3.     int pLen = strlen(p);
4.     next[0] = -1;
5.     int k = -1;
6.     int j = 0;
7.     while (j < pLen - 1)
8.     {
9.         //p[k]表示前缀, p[j]表示后缀
10.        if (k == -1 || p[j] == p[k])
11.        {
12.            ++j;
13.            ++k;
14.            next[j] = k;
15.        }
16.        else
17.        {
18.            k = next[k];
19.        }
20.    }
21. }
```

# KMP算法的时间复杂度分析

---

- 假设现在文本串 $s$ 匹配到 $i$ 位置，模式串 $P$ 匹配到 $j$ 位置，发现如果某个字符匹配成功，模式串首字符的位置保持不动，仅仅是 $i++$ 、 $j++$ ；如果匹配失败， $i$ 不变（即 $i$ 不回溯），模式串会跳过匹配过的 $next[j]$ 个字符。
- 整个算法最坏的情况是，当模式串首字符位于 $i - j$ 的位置时才匹配成功，算法结束。
- 如果文本串的长度为 $n$ ，模式串的长度为 $m$ ，那么匹配过程的时间复杂度为 $O(n)$ ，算上计算 $next$ 的 $O(m)$ 时间，KMP的整体时间复杂度为 $O(m + n)$ 。

## Next数组的优化

- 如果用之前的next数组方法求模式串“abab”的next数组，可得其next数组为-1 0 0 1（0 0 1 2整体右移一位，初值赋为-1），当它跟下图中的文本串去匹配的时候，发现b跟c失配，于是模式串右移 $j - \text{next}[j] = 3 - 1 = 2$ 位。

a	b	a	c	a	b	a	b	c
---	---	---	---	---	---	---	---	---

a	b	a	b
---	---	---	---

-1   0   0   1

a	b	a	c	a	b	a	b	c
---	---	---	---	---	---	---	---	---

a	b	a	b
---	---	---	---

-1   0   0   1

## Next数组的优化

---

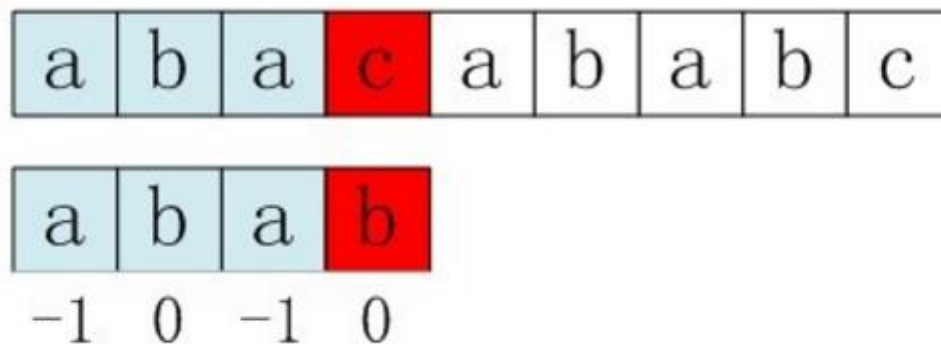
- 右移2位后，b又跟c失配。事实上，因为在上一步的匹配中，已经得知 $p[3] = b$ ，与 $s[3] = c$ 失配，而右移两位之后，让 $p[\text{next}[3]] = p[1] = b$ 再跟 $s[3]$ 匹配时，必然失配。问题出在哪呢？
- **问题出在不该出现 $p[j] = p[\text{next}[j]]$ ]**
- **当 $p[j] \neq s[i]$ 时，下次匹配必然是 $p[\text{next}[j]]$ 跟 $s[i]$ 匹配，如果 $p[j] = p[\text{next}[j]]$ ，必然导致后一步匹配失败，所以不能允许 $p[j] = p[\text{next}[j]]$**

# Next数组的优化

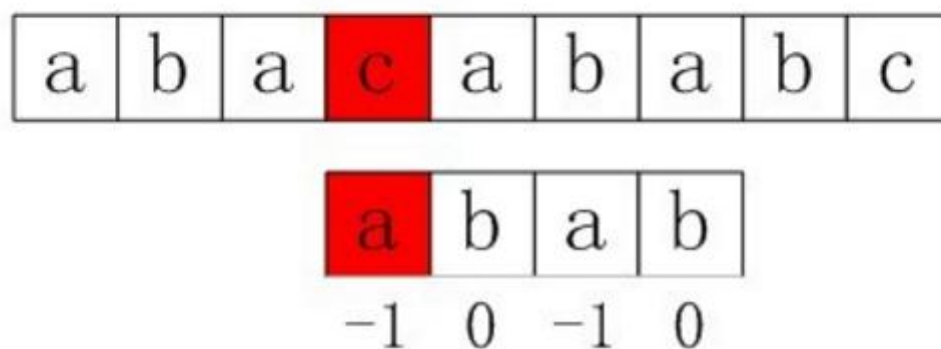
模式串	a	b	a	b
最大长度值	0	0	1	2
未优化next数组	next[0] = -1	next[1] = 0	next[2] = 0	next[3] = 1
索引值	$p_0$	$p_1$	$p_2$	$p_3$
优化理由	初值不变	$p[1] \neq p[\text{next}[1]]$	因 $p_j$ 不能等于 $p[\text{next}[j]]$ ，即 $p[2]$ 不能等于 $p[\text{next}[2]]$	$p[3]$ 不能等于 $p[\text{next}[3]]$
措施	无需处理	无需处理	$\text{next}[2] = \text{next}[\text{next}[2]] = \text{next}[0] = -1$	$\text{next}[3] = \text{next}[\text{next}[3]] = \text{next}[1] = 0$
优化的next数组	-1	0	-1	0

# Next数组的优化

1.  $S[3]$ 与 $P[3]$ 匹配失败。

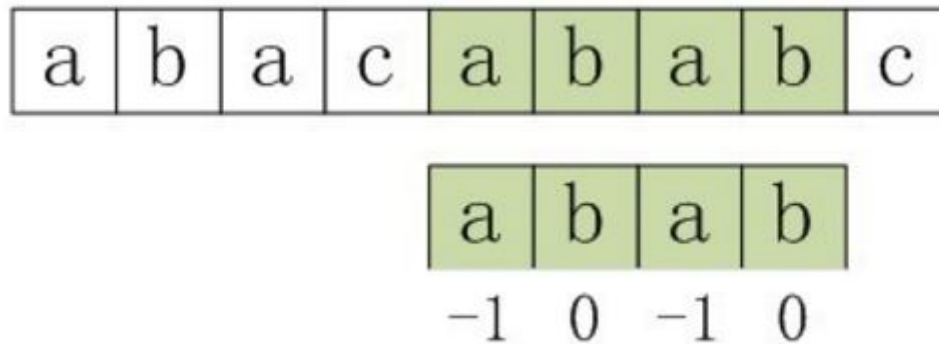


2.  $S[3]$ 保持不变， $P$ 的下一个匹配位置是 $P[\text{next}[3]]$ ，而 $\text{next}[3]=0$ ，所以 $P[\text{next}[3]]=P[0]$ 与 $S[3]$ 匹配。



## Next数组的优化

- 由于上一步骤中P[0]与S[3]还是不匹配。此时 $i=3$ ， $j=\text{next}[0]=-1$ ，由于满足条件 $j=-1$ ，所以执行“ $++i, ++j$ ”，即主串指针下移一个位置，P[0]与S[4]开始匹配。最后 $j==\text{pLen}$ ，跳出循环，输出结果 $i - j = 4$ （即模式串第一次在文本串中出现的位置），匹配成功，算法结束。



---

# 谢谢！

