4 字符串

董洪伟

http://hwdong.com

主要内容

- 串的类型定义
- 串的表示和实现
  - 单个字符数组表示
  - 结构表示法
  - 块链存储
- 串的模式匹配
  - 简单算法
  - **KMP**算法

串的类型定义

- 定义
  - 串：n个字符的有限序列，n>=0
    - 比如：S = ‘a_1 a_2. . . a_n’
    - S是串名
    - ‘a_1 a_2. . . a_n’是串S的值
  - 串的长度：串中字符的个数
    - 空串 ∅ ：长度为0的串
    - 注意：空串≠空格串

串的类型定义

- 子串：串中任意多个连续字符组成的子序列
- 位置：字符在序列中的序号
  - 子串的位置 = 子串第一个字符的位置
- 串相等：两个串的值相等
  - 长度相等
  - 各个对应位置的字符也相同

串的类型定义

ADT String{

逻辑结构：字符序列

基本操作：

```
bool init(String &S,const char *);
void clear(String &S);
int  size(String S); //字符个数
String subStr(String S,int pos,int len);
bool insert(String &S,String T,int pos);
bool cat(String &R, String S, String T);
int find(String S,int pos, String T);
bool copy(String &S,String T);
bool  erase(String S,int pos,int len);
```

顺序串：单个字符数组

- 静态顺序存储

```
char s[10];
```

- 堆分配存储（动态顺序存储）

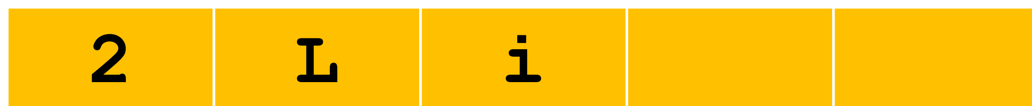
```
char *s=(char *)malloc(10*sizeof(char));
```

- 串长表示法

1）结尾加结束字符:'\0'

L	i	\0		

2）开头存放串长信息



2	L	i		

顺序串：单个字符数组

- 串长表示法：结尾加结束字符:'\0'

L	i	\0		

- 字符数组！=字符串

```
s[0] = 'L';s[1] = 'i';
int len = strlen(s) ;//错!
s[2] = '\0';
len = strlen(s) ; //正确
```

顺序串：单个字符数组

```
int strlen(const char *str){
    int i = 0 ;
    while(str[i]!='\0')i++;
    return i;
}

int strlen(const char *str){
    const char *p =str;
    while(*p!='\0')p++;
    return p-str;
}
```

顺序串：单个字符数组

```
char *strcpy(char *s1, const char *s2){
    char *s = s1;
    while ((*s++ = *s2++) != 0)  ;
    return (s1);
}
```

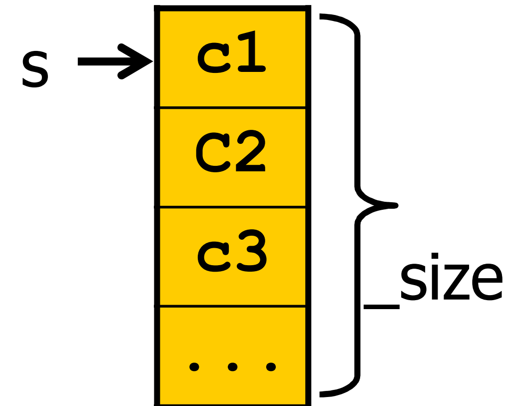
顺序串：静态和动态存储

- **静态分配存储：** 静态的连续空间
- **堆分配存储：** 动态分配的连续空间
- **动态分配的优点：**
  - 既有静态存储的特点
  - 又没有长度限制

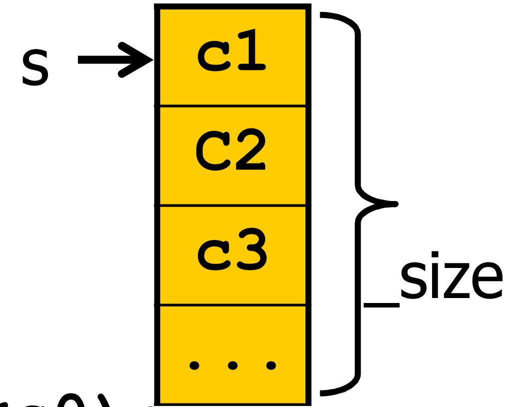
顺序串：结构表示法

- 用辅助变量存放串长：顺序表结构

```
typedef struct{
    char *s;
    int _size;
}String;
```

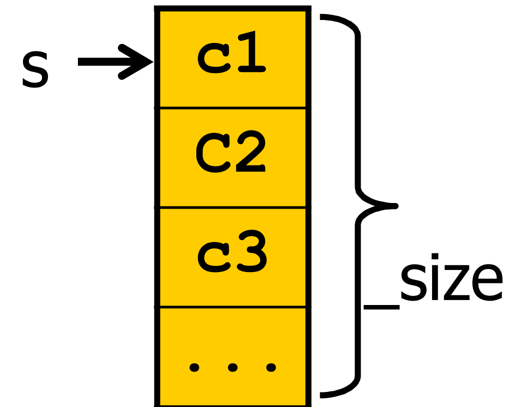


顺序串：结构表示法




```
typedef struct{
    char *s;
    int _size;
}String;
bool initString(String &s,char *s0);
void destoryString(String &s);
void clearString(String &s);
int size( String s);
int catString(String &T,String s1,String s2);
String subString(String S,int pos,int len);
int findString(String S,String T,int pos);
int insertString(String &S,String T,int pos);
```

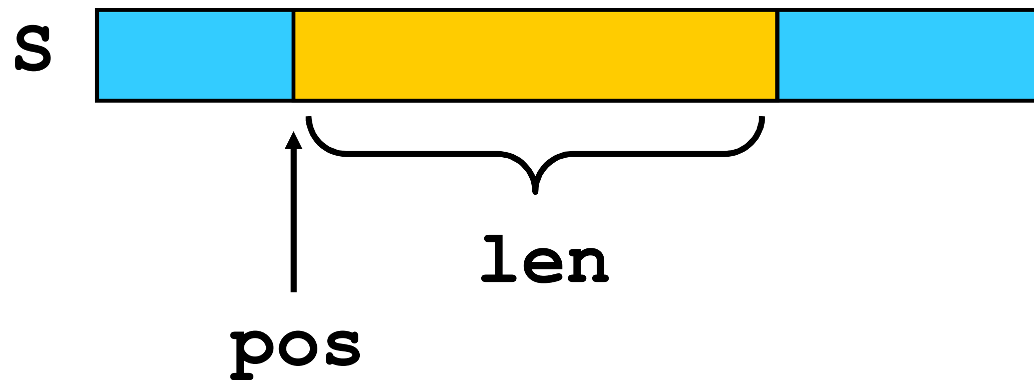
顺序串：结构表示法




```
typedef struct{
    char *s;
    int _size;
}String;
bool initString(String &S,char *s0){
    int len = strlen(s0);
    S.s = (char *)malloc((len+1)*sizeof(char));
    if(!S.s) return false;
    strcpy(S.s,s0);
    S._size = len;
    return true;
}
```

顺序串：结构表示法

- 求子串
  - 串S第pos个字符起，长度为len的子串

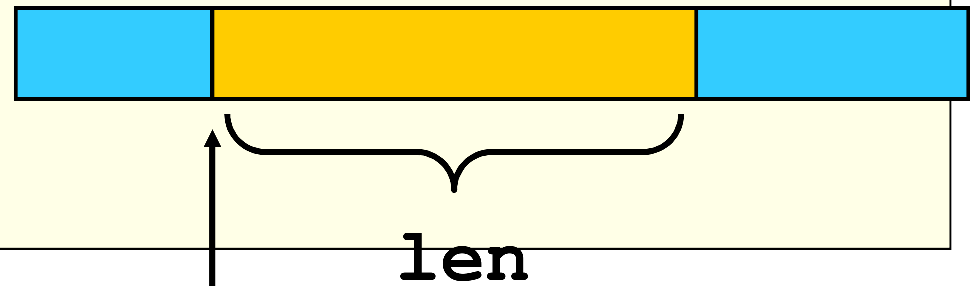



```
S = "abvdfkhdsfg";
subS = subString(S,2,3);
```

顺序串：结构表示法

- 算法

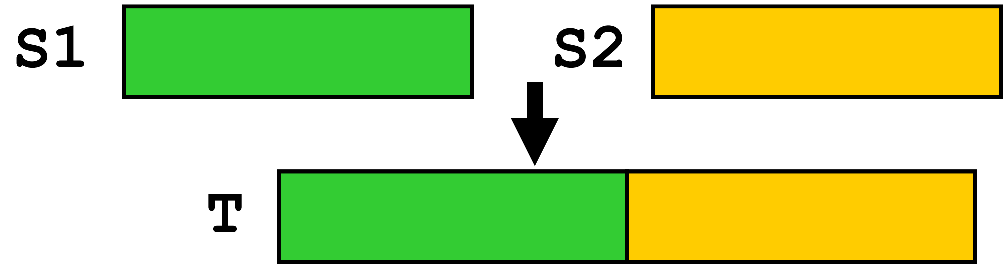
```
String subString(String S,int pos,int len)
{
   String T; T.s = 0;T._size = 0;
   T.s = (char*)malloc((len+1)*sizeof(char));
   if(T.s){
      for(int i = 0 ;i<len;i++)
         T.s[i] = S.s[i+pos-1];
      T.s[len] = '\0';
      T._size = len;
   }
   return T;
}
```



Any Bug?

顺序串：结构表示法

- 串的拼接




```
int catString(String &T,String s1,String s2)
{
  int len = s1._size+s2._size;
  T.s = (char *)malloc((len+1)*sizeof(char));
  if(!T.s) return 0;
  T._size = len;
  strcpy(T.s,s1.s);
  char *p = T.s+s1._size;
  strcpy(p,s2.s);
  return T._size;
}
```

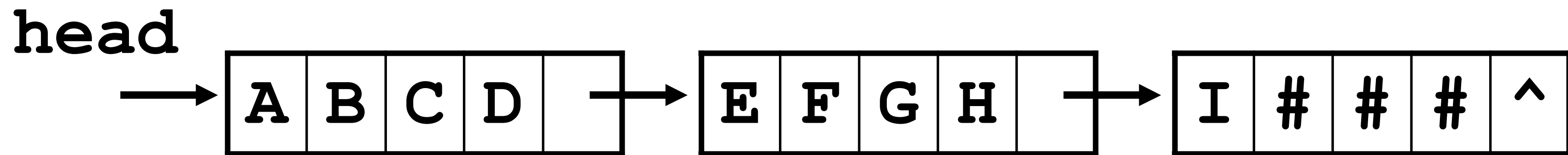
顺序串：结构表示法

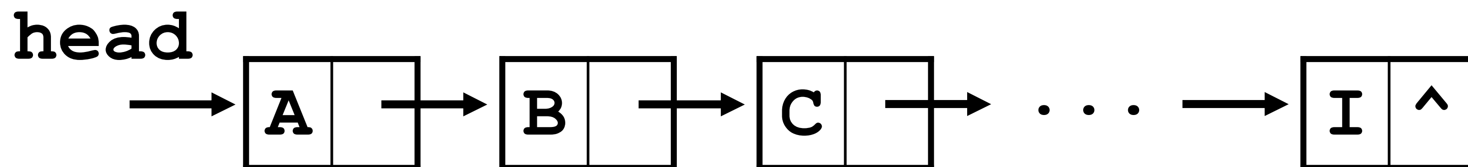
- 插入

```
int insertString (String &S,String T,int pos)
{//1. 插入位置合法？

 //2. 分配空间

 //3. 拷贝数据

}
```

链式串：块链存储

- 块链存储：用链表存储字符串





串的表示和实现：块链存储

- 优点
  - 便于拼接操作
- 缺点
  - 结点大小需要设置恰当
  - 存储密度 $= \dfrac{\text{串值所占空间}}{\text{实际分配空间}}$
  - 结点越小，存储密度越小，操作越方便，但是存储空间浪费大

串的模式匹配

- 模式匹配
  - 在主串**S**中定位子串**T**（模式串）
  - 回忆一下串匹配的定义：
    - **find(S, T, pos)**
    - 初始条件：串**S**和**T**存在，**T**非空，**1 <= pos <= StrLength(S)**
    - 操作结果：若主串**S**中存在和串**T**值相同的子串，返回它在主串**S**中第**pos**个字符之后第一次出现的位置；否则返回**0**

串的模式匹配

– 例如

- 主串S =

1	2	3	4	5	6	7	8
A	B	C	D	E	C	D	H

- 子串T = CD
- 则find(S, T, 2), 返回从位置2起，子串T在S中，第一次出现的位置3

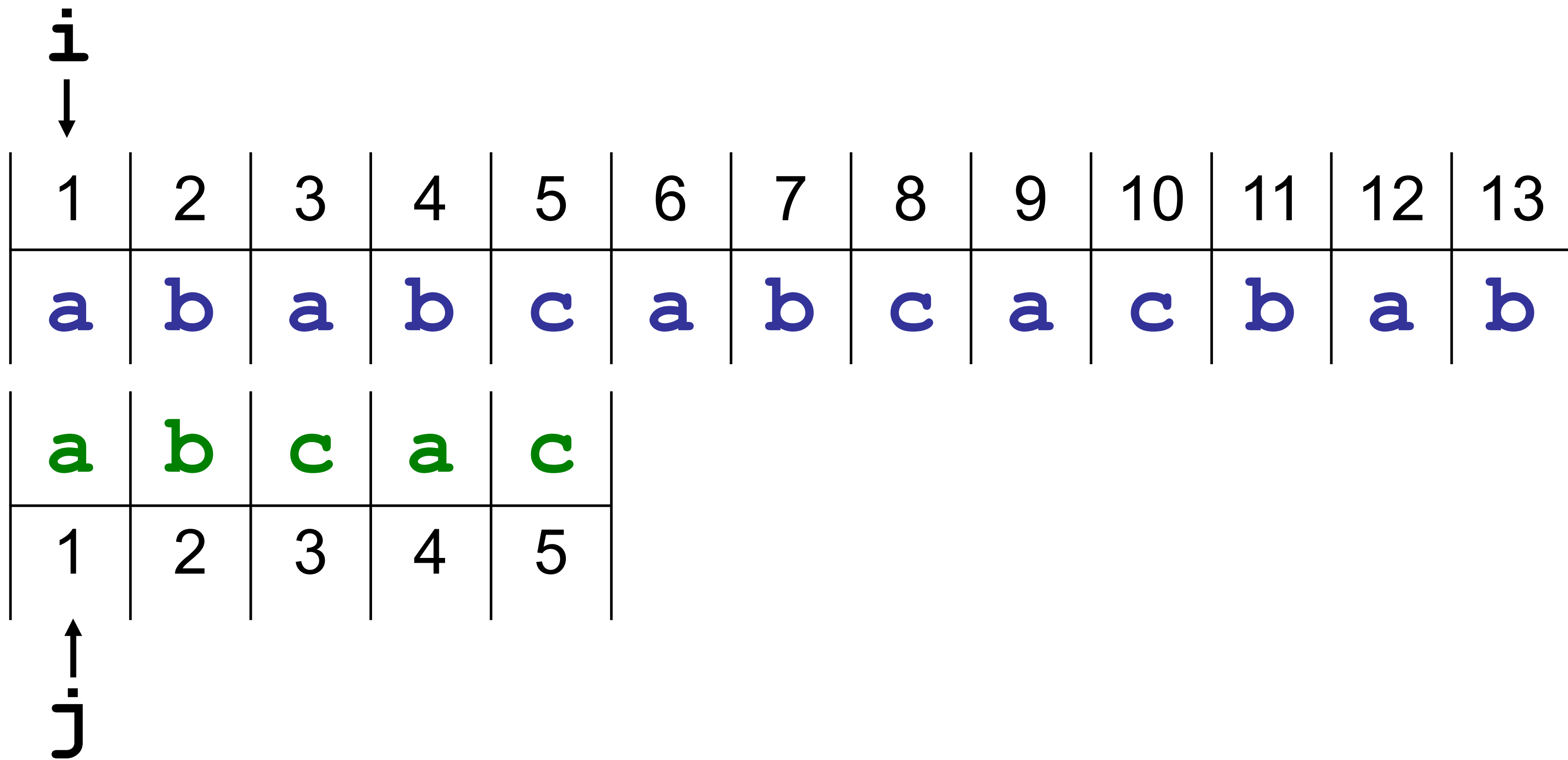
串的模式匹配

- 以定长顺序表示时的几种算法
  - 简单算法
  - **KMP**算法(**D.E.Knuth， J.H.Morris, V.R.Pratt)**

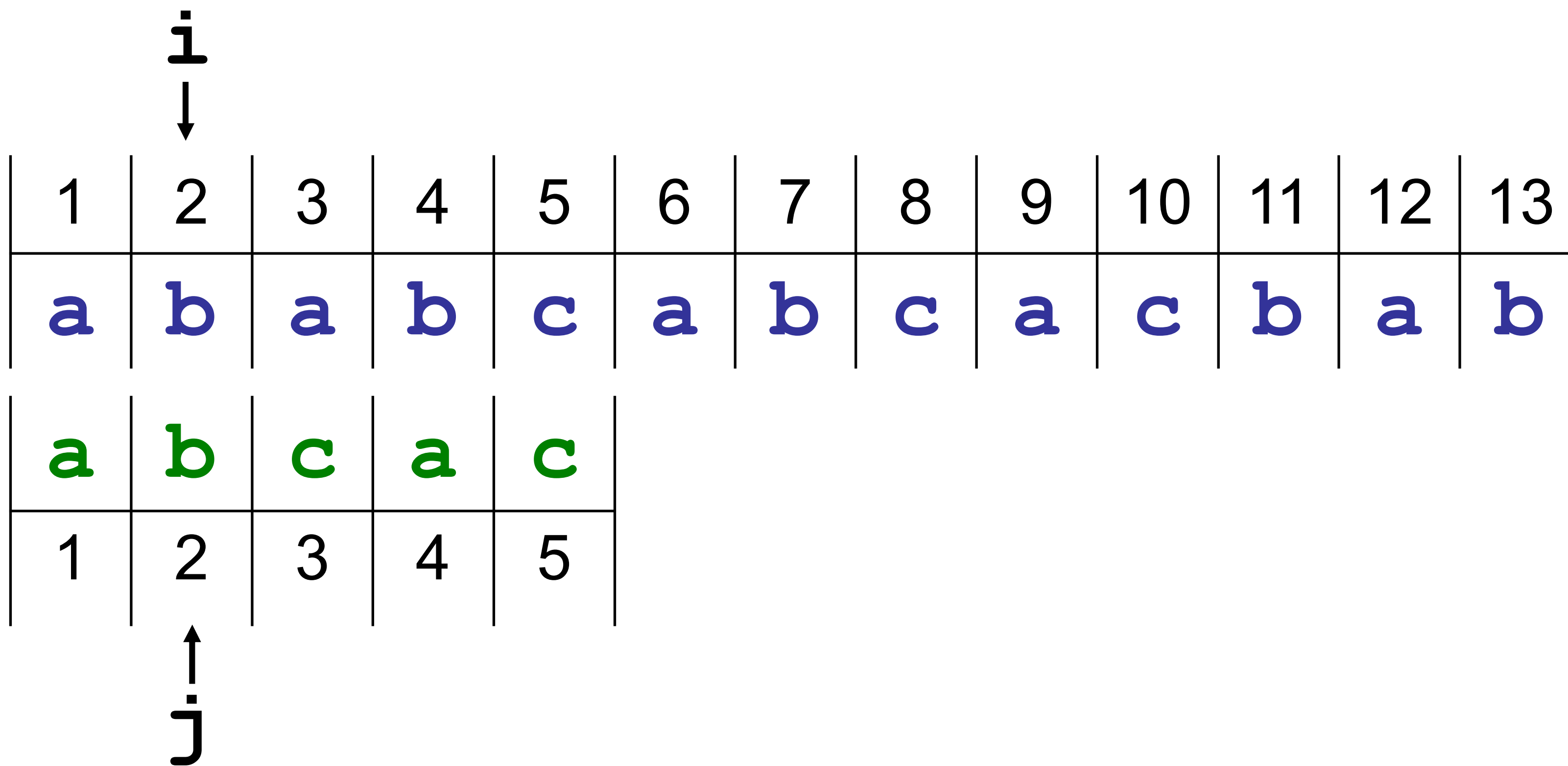
- 简单算法

```
int find(String S, String T, int pos){
   int i = pos-1,j = 0;
   while(i < S._size && j < T._size ){
      if(S.s[i] == T.s[j]){//当前字符匹配, i,j递增
         i++;  j++;
      }
      else { //否则i回退, j返回模式串首, 重新开始
         i = i - j + 1;  j = 0;
      }
   }
   if(j >= T._size) return i-j+1; //匹配成功
   else             return 0;         //失败
}
```

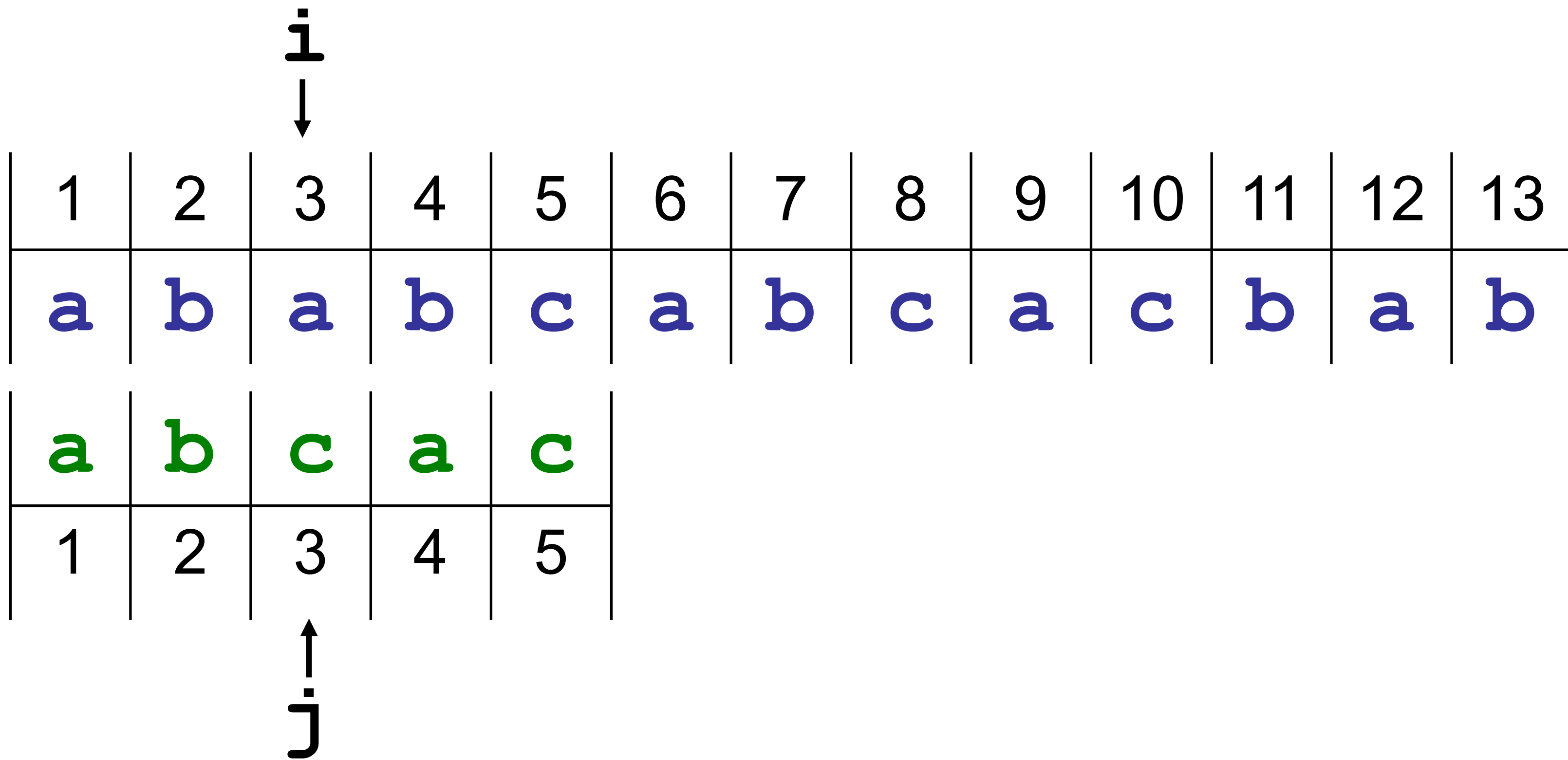
```
while(i < S._size && j < T._size ){
   if(S.s[i] == T.s[j]){ //当前字符匹配, i,j递增
       i++;  j++; }
   else { //匹配失败, j返回子串首, i回退到下次匹配的起点
       i = i - j + 1;  j = 0;  }}
```



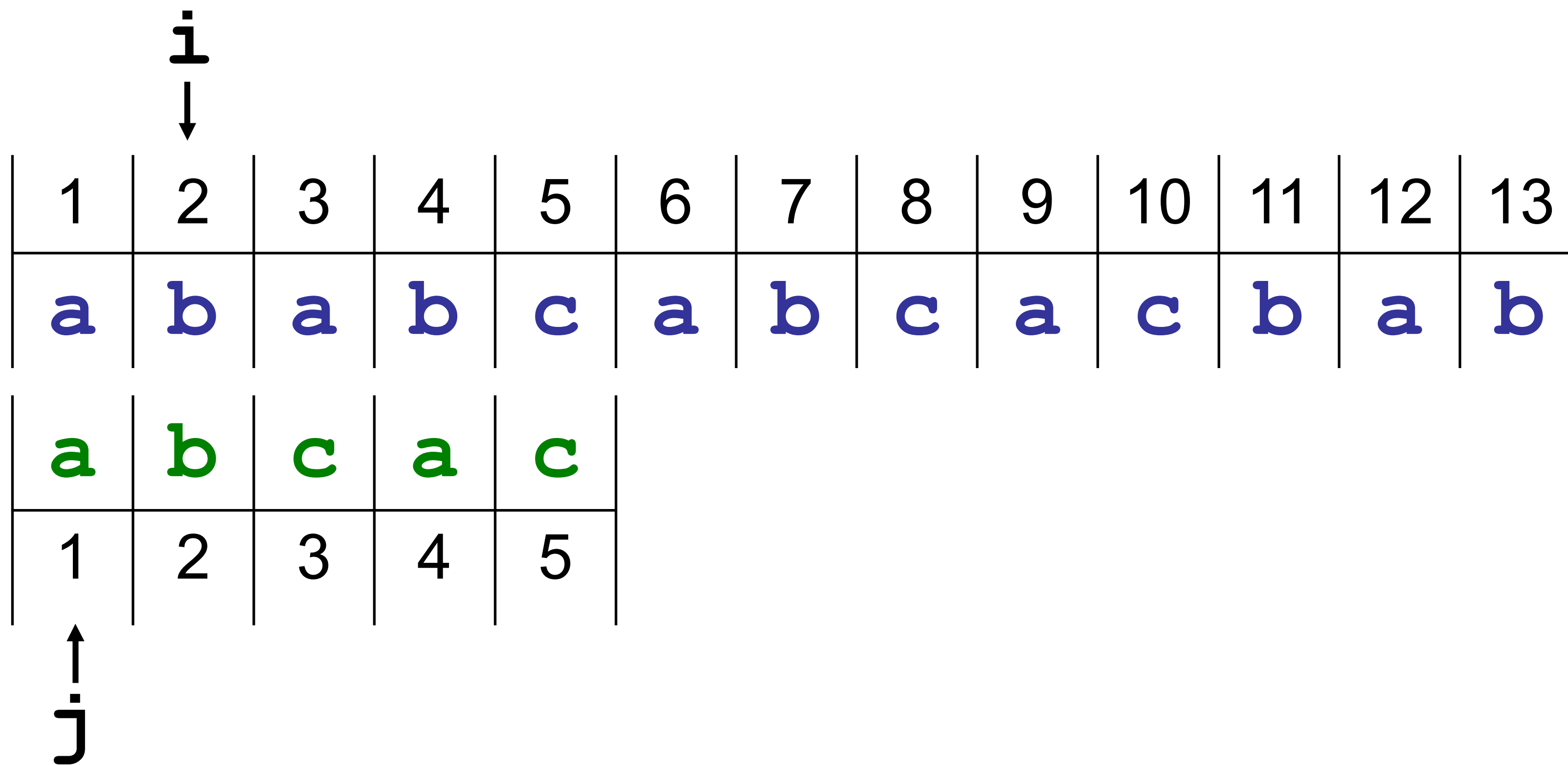
```
while(i < S._size && j < T._size ){
   if(S.s[i] == T.s[j]){ //当前字符匹配，i,j递增
       i++;  j++; }
   else { //匹配失败，j返回子串首，i回退到下次匹配的起点
       i = i - j + 1;  j = 0;  }}
```



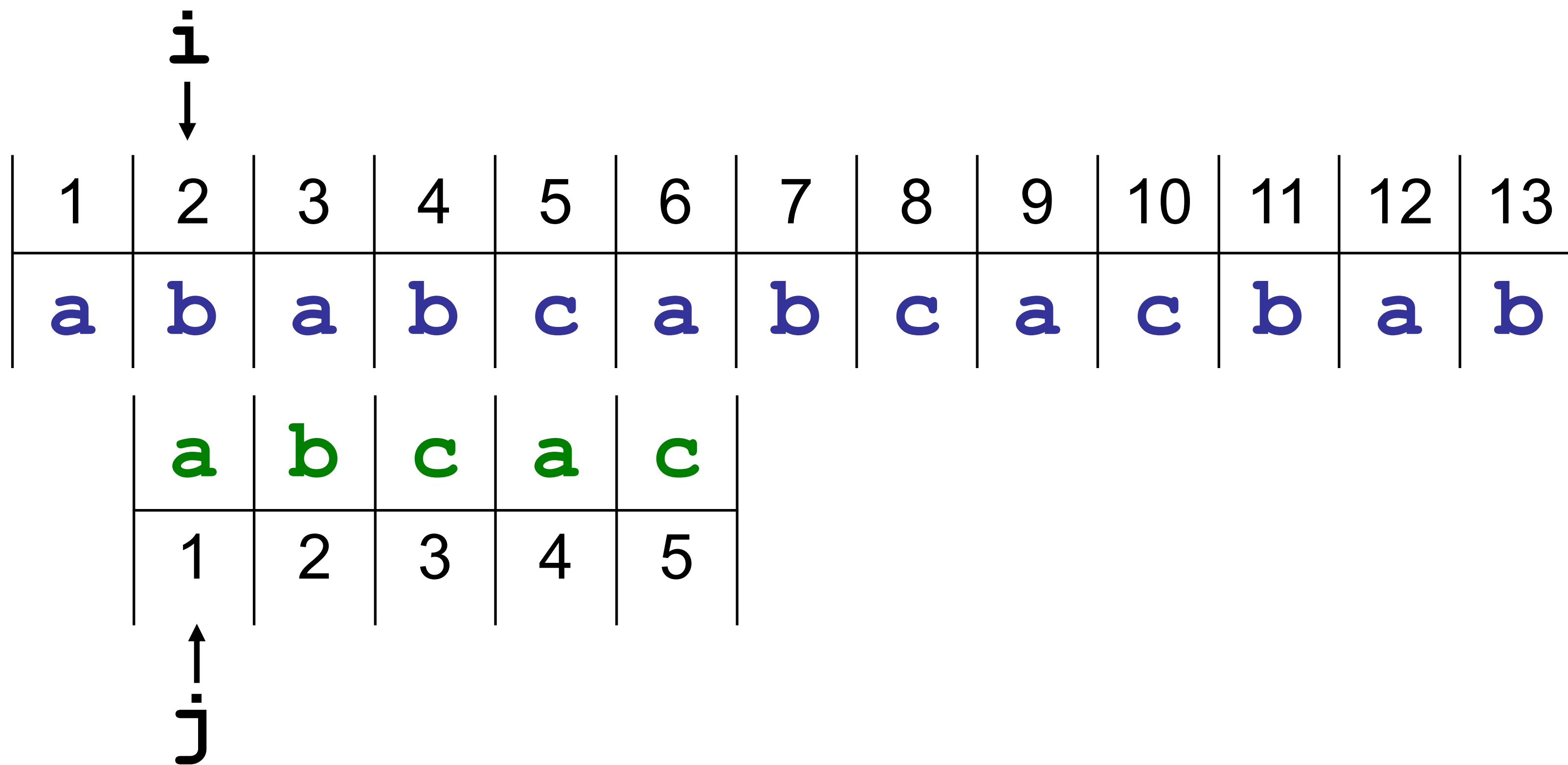
```
while(i < S._size && j < T._size ){
   if(S.s[i] == T.s[j]){ //当前字符匹配，i,j递增
       i++;  j++; }
   else { //匹配失败，j返回子串首，i回退到下次匹配的起点
       i = i - j + 1;  j = 0;  }}
```



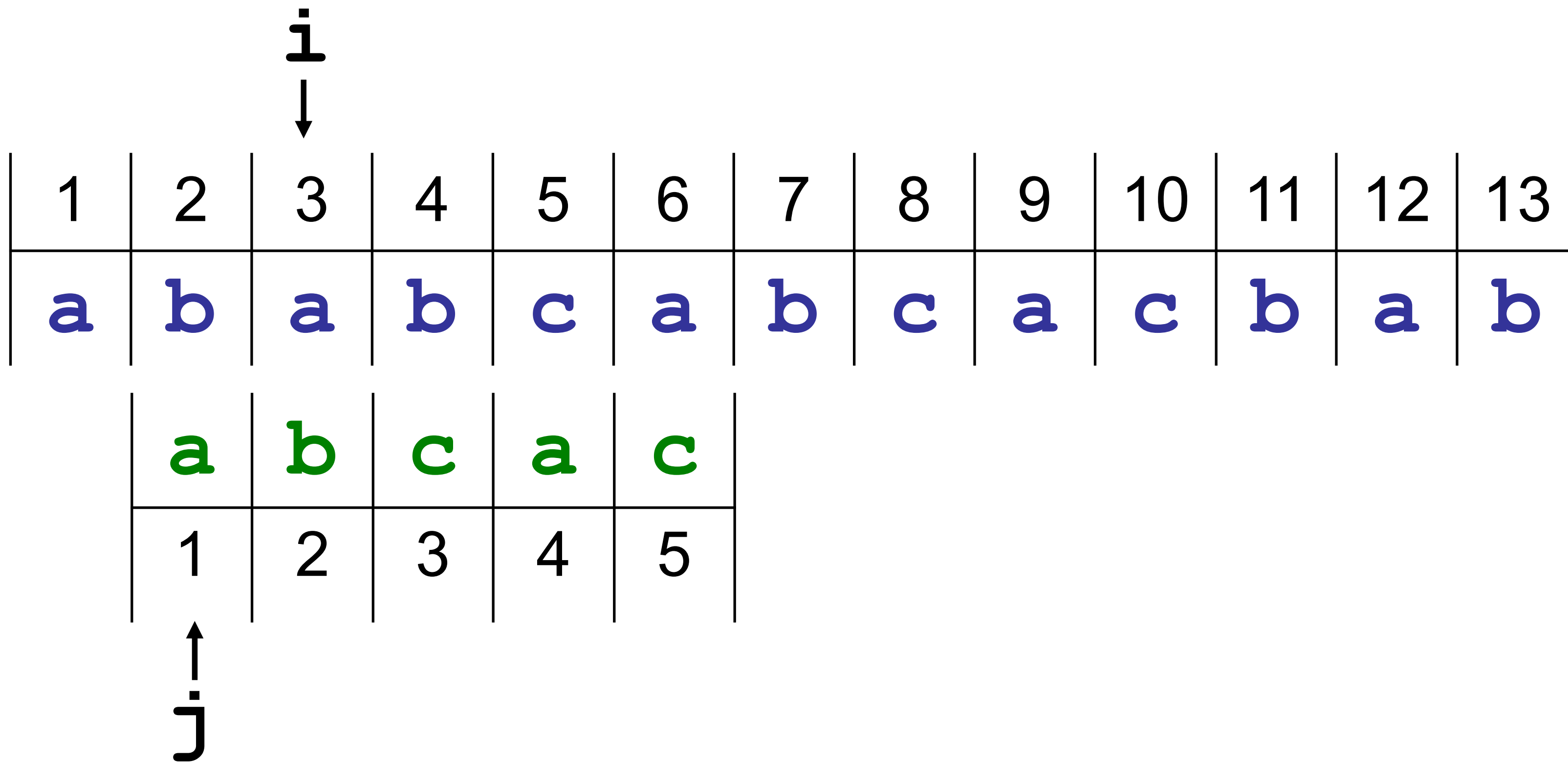
```
while(i < S._size && j < T._size ){
   if(S.s[i] == T.s[j]){ //当前字符匹配, i,j递增
       i++;  j++; }
   else { //匹配失败, j返回子串首, i回退到下次匹配的起点
       i = i - j + 1;  j = 0;  }}
```



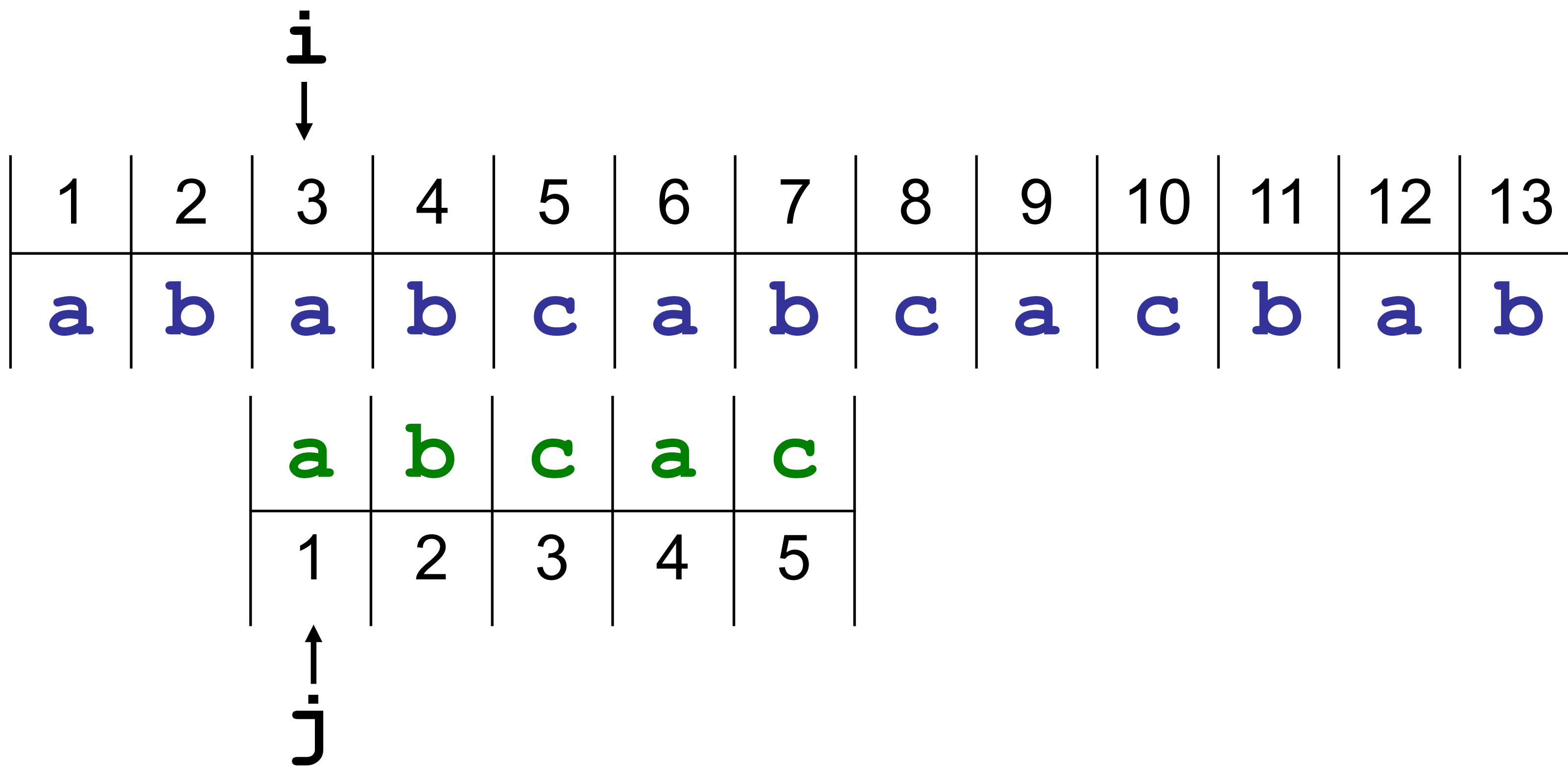
```
while(i < S._size && j < T._size ){
   if(S.s[i] == T.s[j]){ //当前字符匹配, i,j递增
       i++;  j++; }
   else { //匹配失败, j返回子串首, i回退到下次匹配的起点
       i = i - j + 1;  j = 0;  }}
```



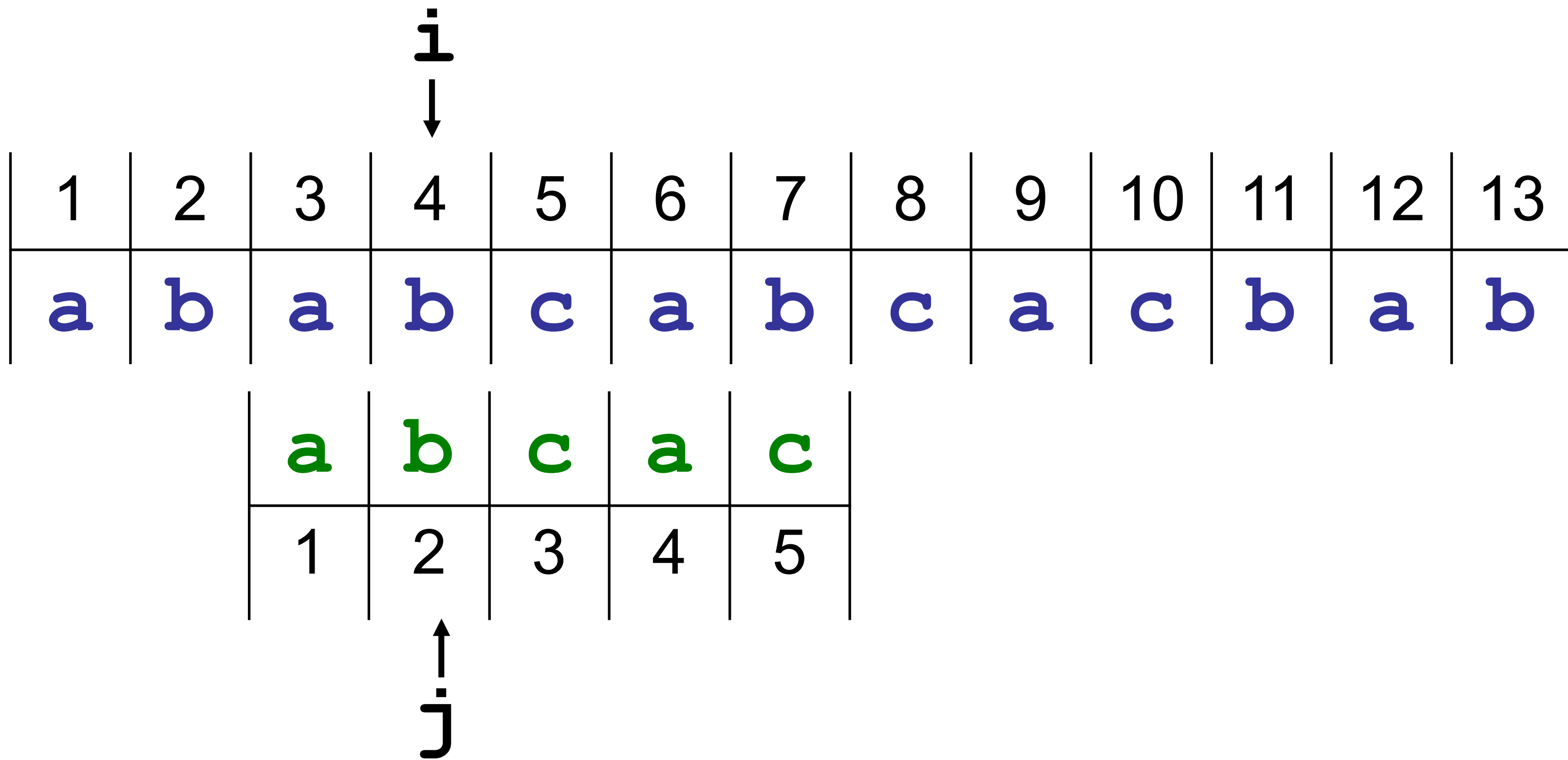
```
while(i < S._size && j < T._size ){
   if(S.s[i] == T.s[j]){ //当前字符匹配，i,j递增
        i++;  j++; }
   else { //匹配失败，j返回子串首，i回退到下次匹配的起点
        i = i - j + 1;  j = 0;  }}
```



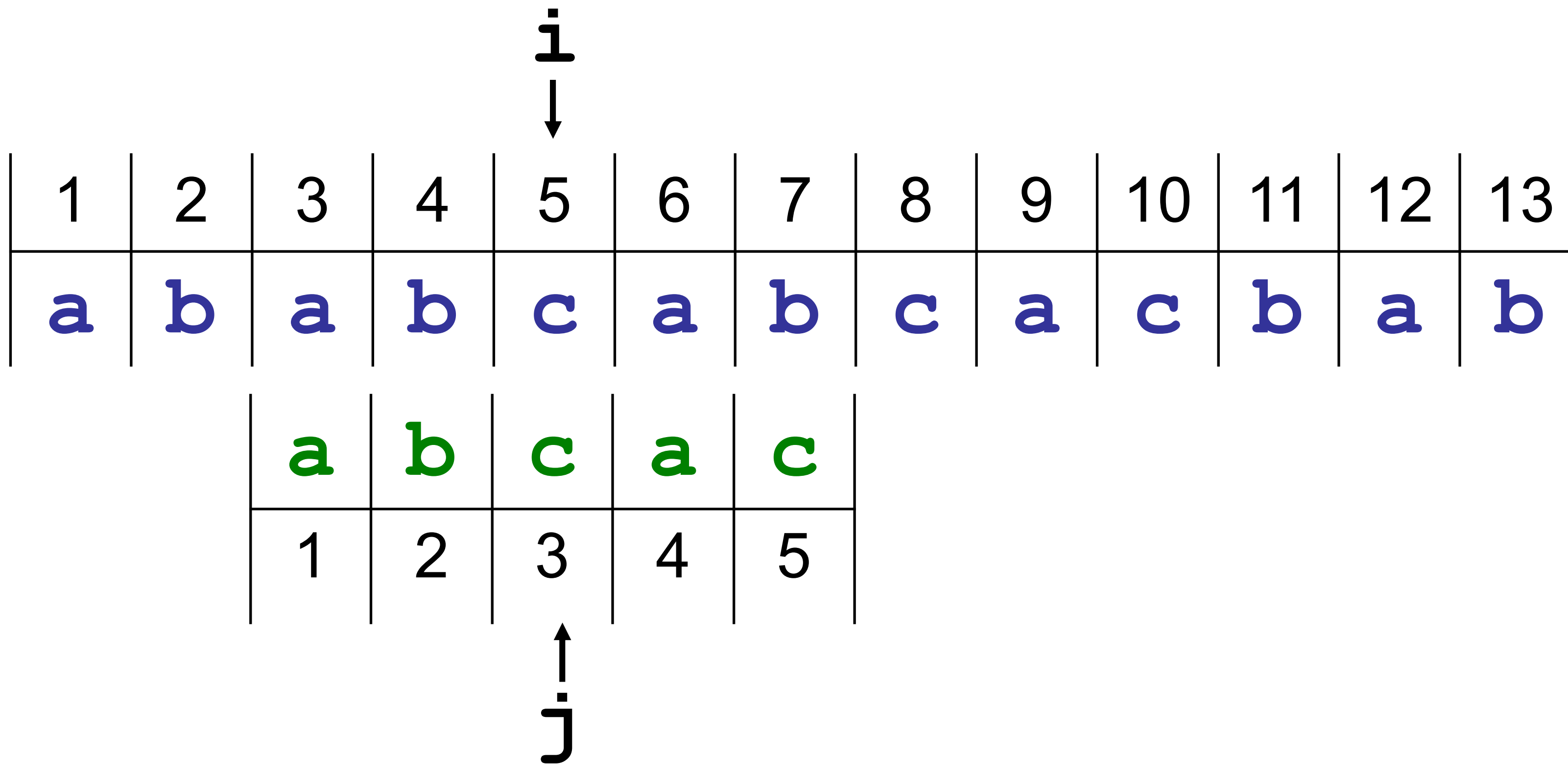
```
while(i < S._size && j < T._size ){
   if(S.s[i] == T.s[j]){ //当前字符匹配, i,j递增
        i++;  j++; }
   else { //匹配失败, j返回子串首, i回退到下次匹配的起点
        i = i - j + 1;  j = 0;  }}
```



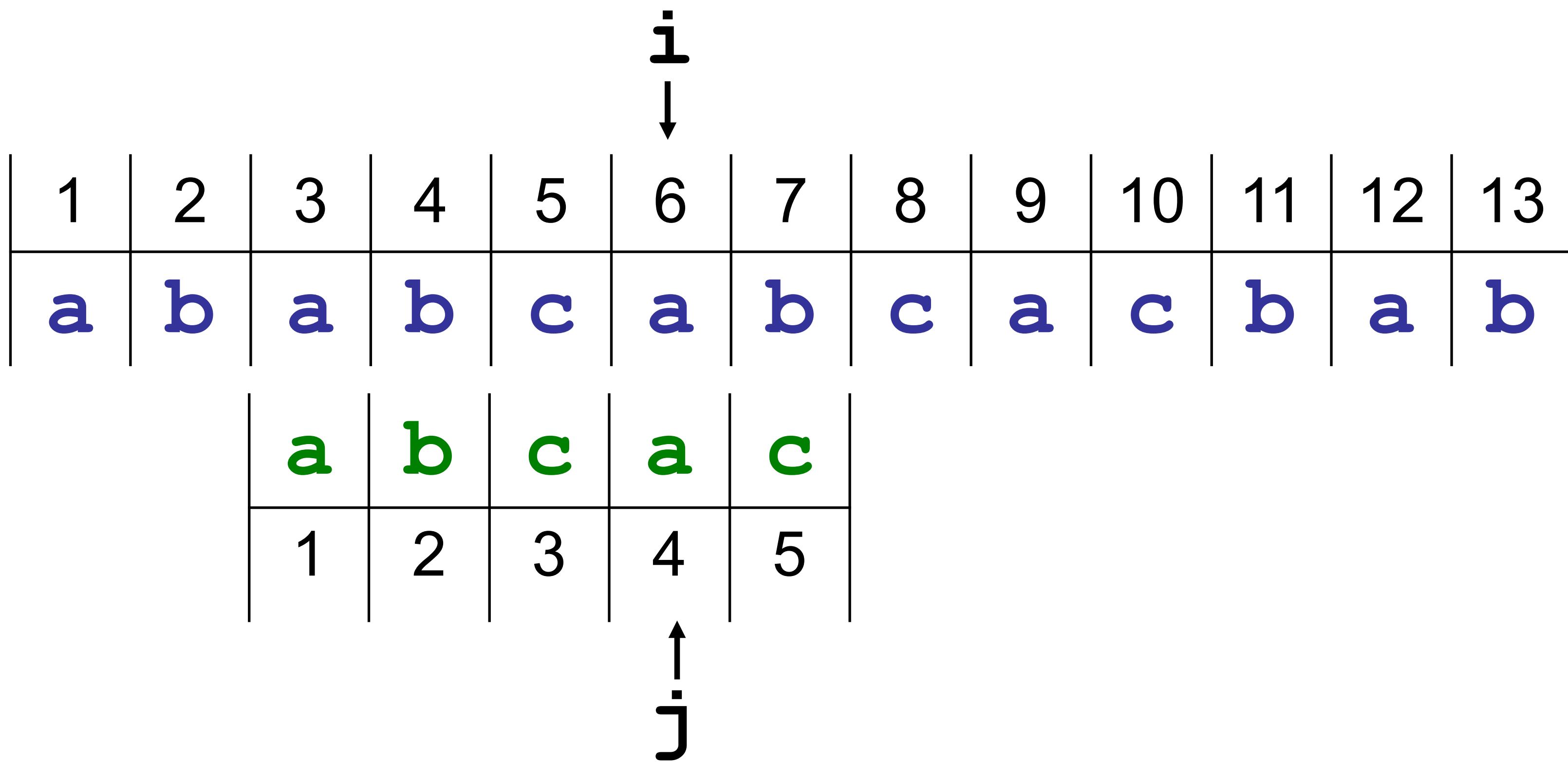
```
while(i < S._size && j < T._size ){
   if(S.s[i] == T.s[j]){ //当前字符匹配，i,j递增
       i++;  j++; }
   else { //匹配失败，j返回子串首，i回退到下次匹配的起点
       i = i - j + 1;  j = 0;  }}
```



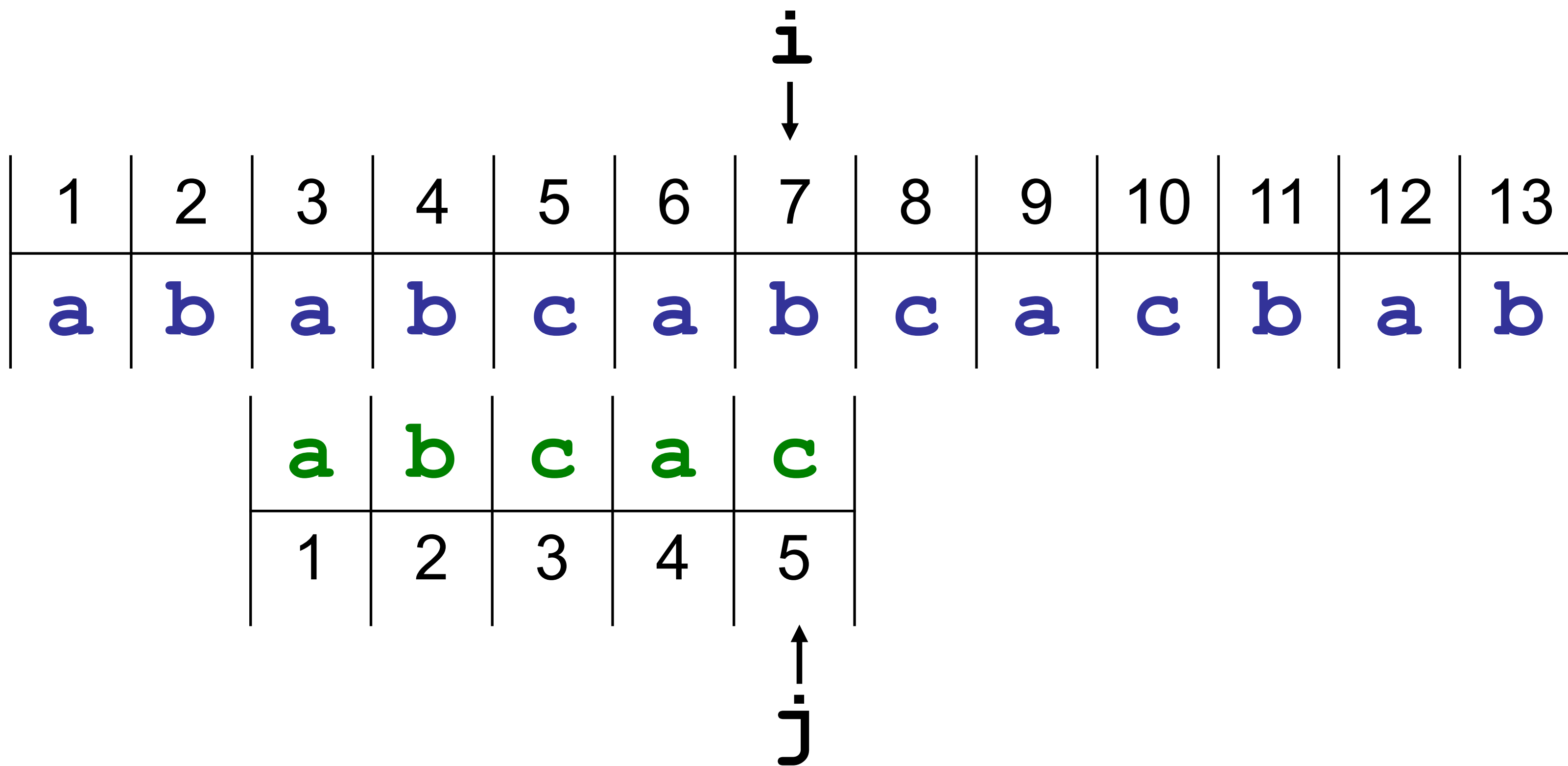
```
while(i < S._size && j < T._size ){
   if(S.s[i] == T.s[j]){ //当前字符匹配, i,j递增
       i++;  j++; }
   else { //匹配失败, j返回子串首, i回退到下次匹配的起点
       i = i - j + 1;  j = 0;  }}
```



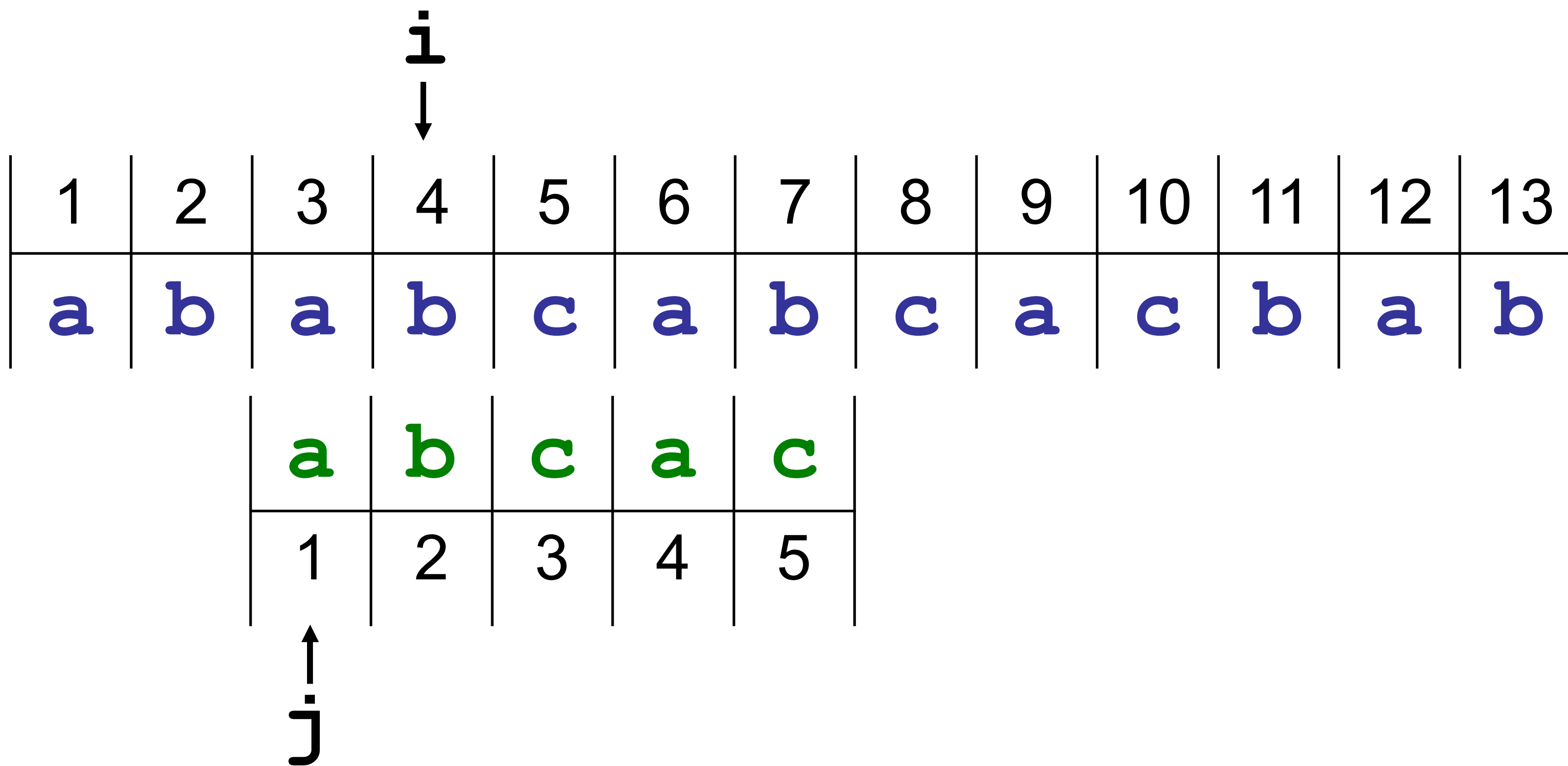
```
while(i < S._size && j < T._size ){
   if(S.s[i] == T.s[j]){ //当前字符匹配, i,j递增
        i++;  j++; }
   else { //匹配失败, j返回子串首, i回退到下次匹配的起点
        i = i - j + 1;  j = 0;  }}
```



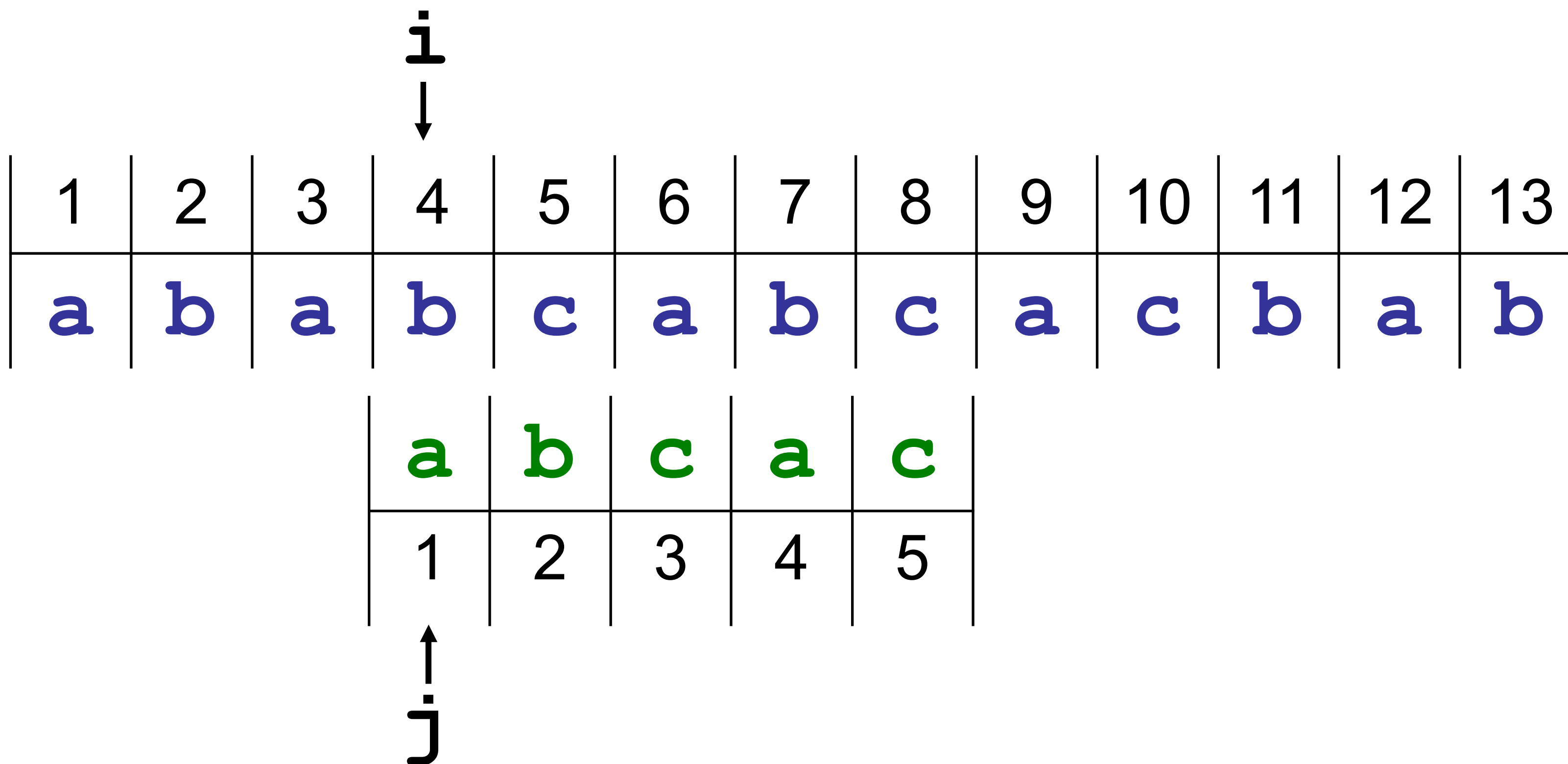
```
while(i < S._size && j < T._size ){
   if(S.s[i] == T.s[j]){ //当前字符匹配，i,j递增
      i++;  j++; }
   else { //匹配失败，j返回子串首，i回退到下次匹配的起点
      i = i - j + 1;  j = 0;  }}
```



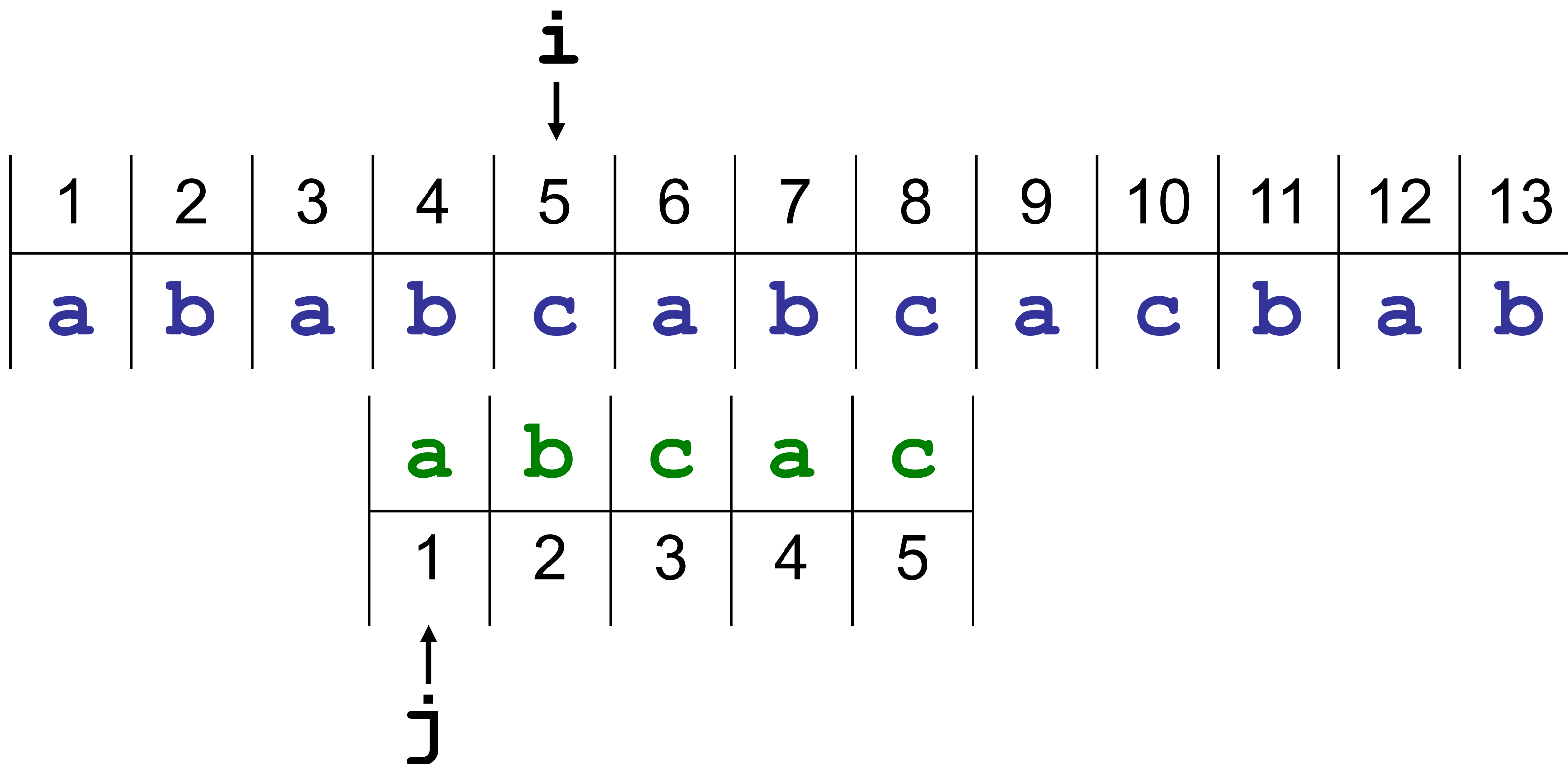
```
while(i < S._size && j < T._size ){
   if(S.s[i] == T.s[j]){ //当前字符匹配，i,j递增
       i++;  j++; }
   else { //匹配失败，j返回子串首，i回退到下次匹配的起点
       i = i - j + 1;  j = 0;  }}
```



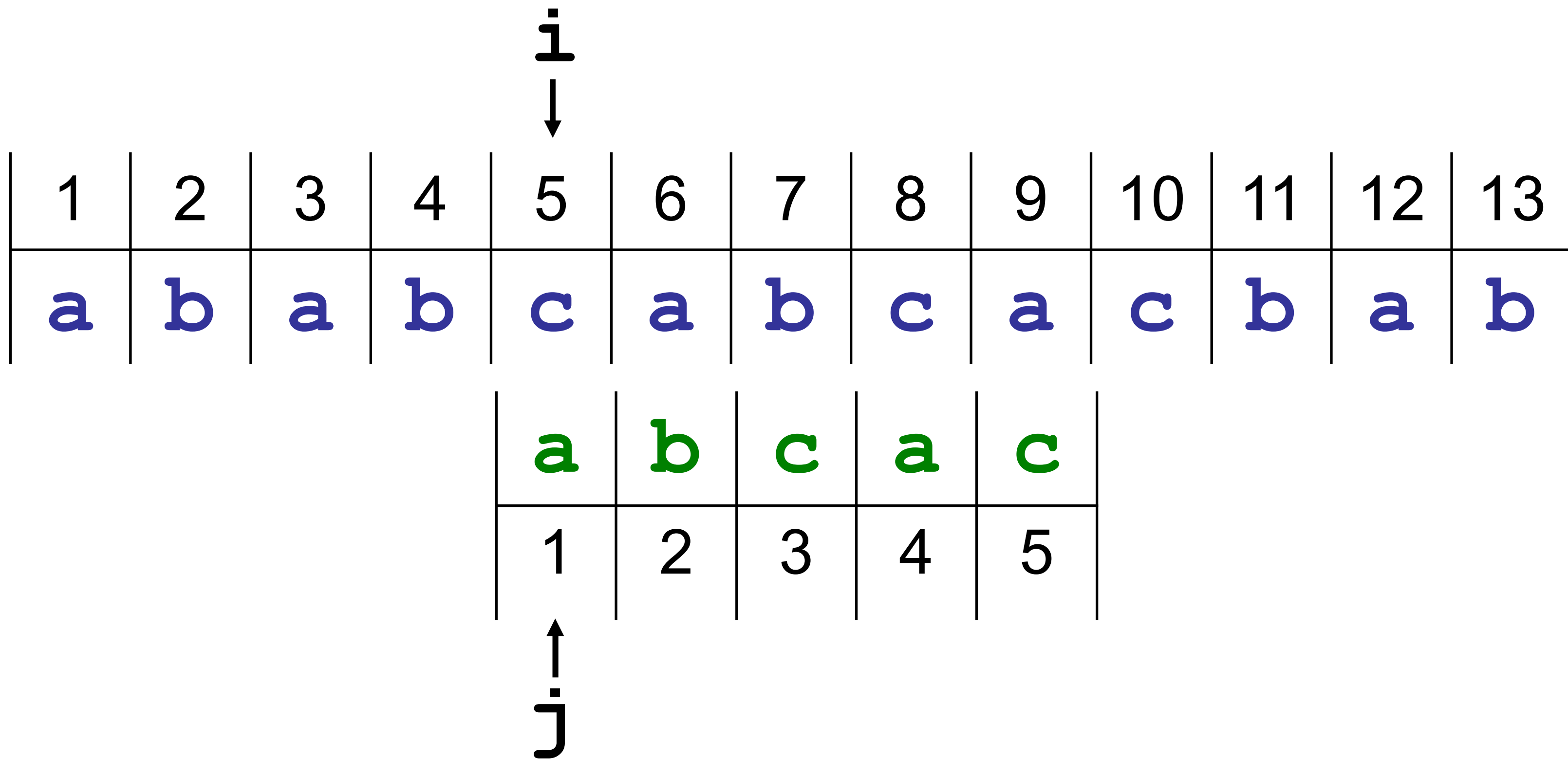
```
while(i < S._size && j < T._size ){
   if(S.s[i] == T.s[j]){ //当前字符匹配，i,j递增
       i++;  j++; }
   else { //匹配失败，j返回子串首，i回退到下次匹配的起点
       i = i - j + 1;  j = 0;  }}
```



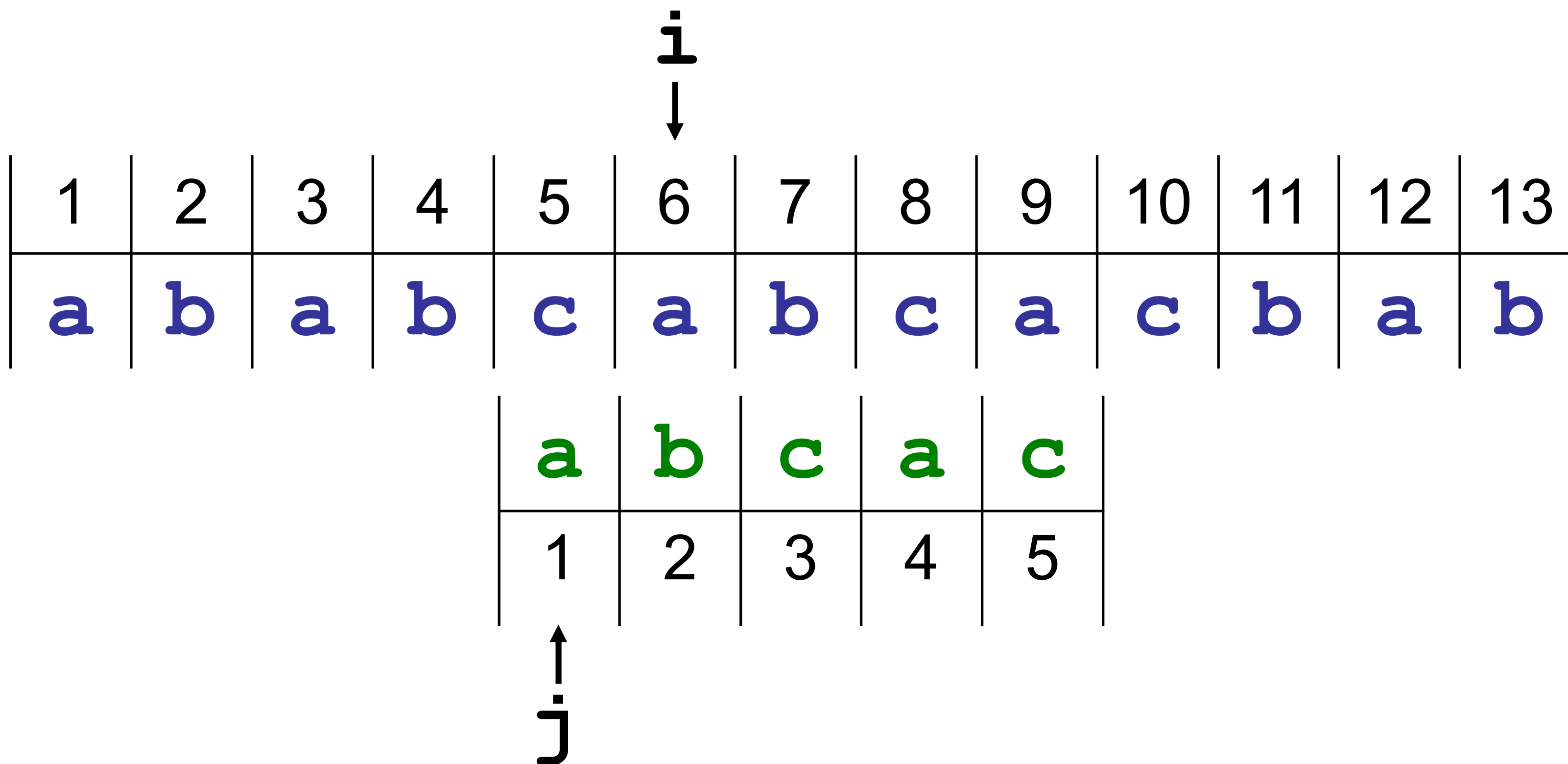
```
while(i < S._size && j < T._size ){
   if(S.s[i] == T.s[j]){ //当前字符匹配，i,j递增
       i++;  j++; }
   else { //匹配失败，j返回子串首，i回退到下次匹配的起点
       i = i - j + 1;  j = 0;  }}
```



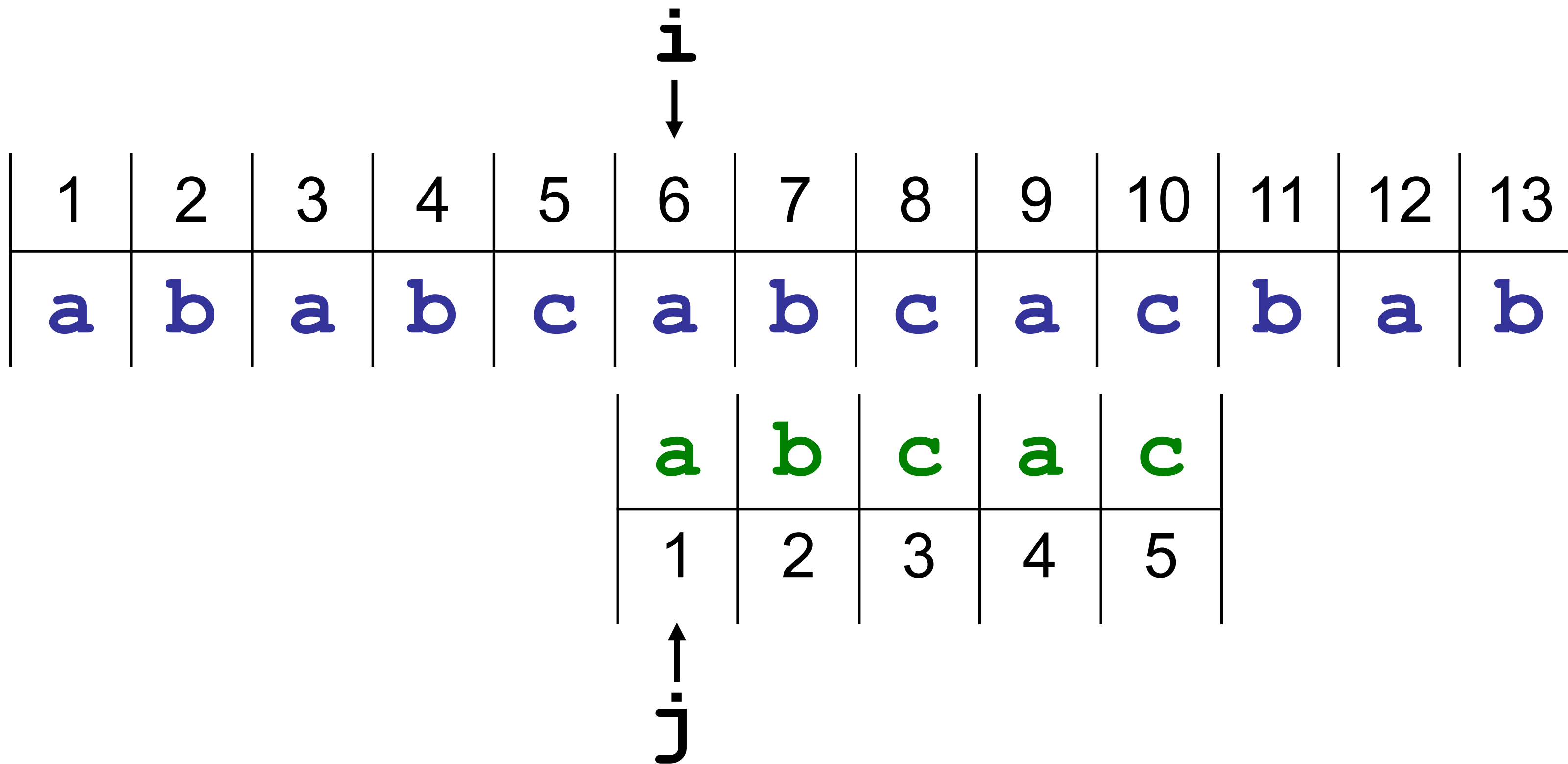
```
while(i <= S[0] && j <= T[0]){
   if(S[i] == T[j]){ //当前字符匹配，i,j递增
       i ++;  j ++; }
   else { //匹配失败，j返回子串首，i回退到下次匹配的起点
       i = i - j + 1;  j = 1;  }}
```



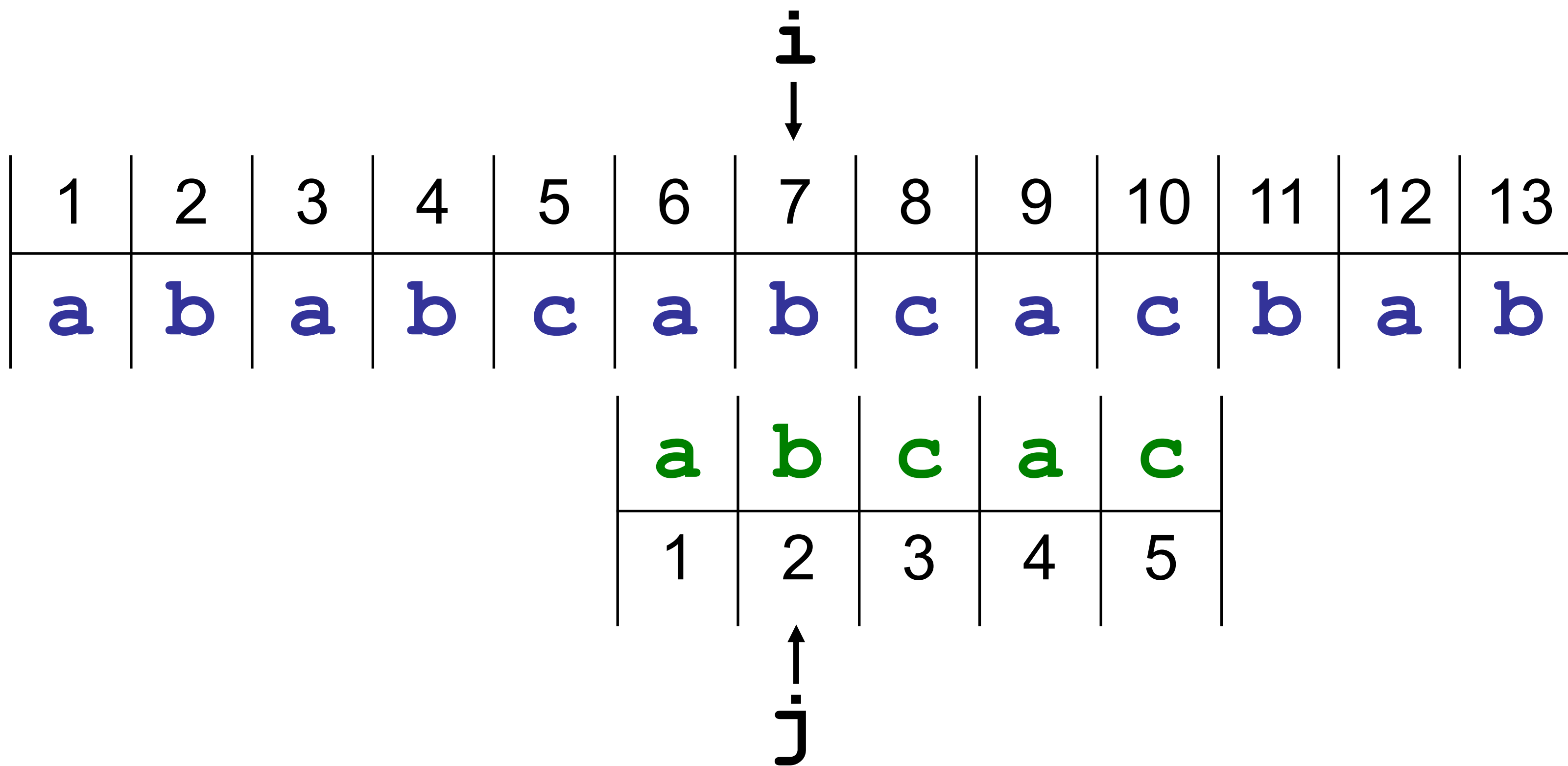
```
while(i < S._size && j < T._size ){
   if(S.s[i] == T.s[j]){ //当前字符匹配，i,j递增
       i++;  j++; }
   else { //匹配失败，j返回子串首，i回退到下次匹配的起点
       i = i - j + 1;  j = 0;  }}
```



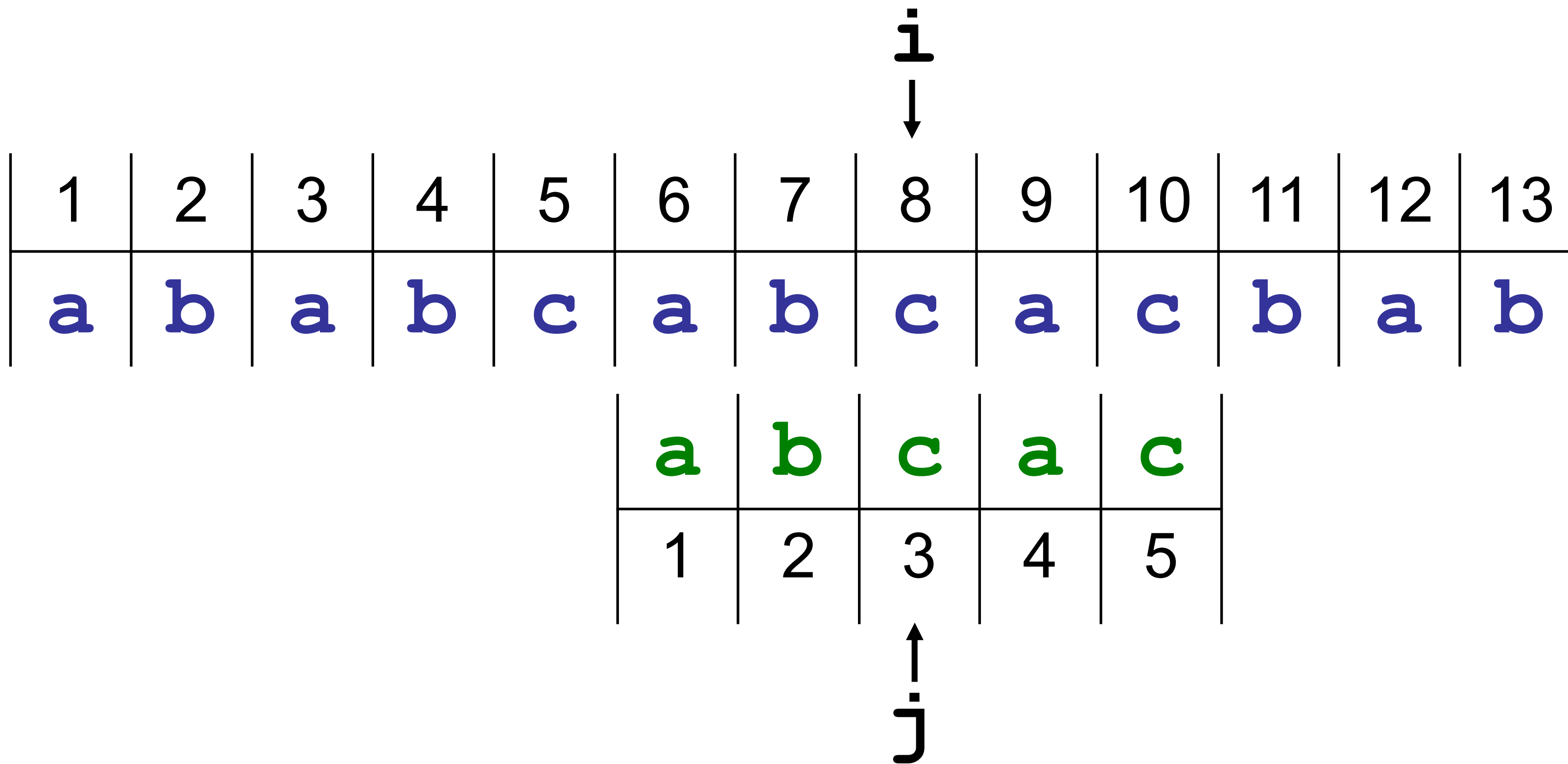
```
while(i < S._size && j < T._size ){
   if(S.s[i] == T.s[j]){ //当前字符匹配，i,j递增
        i++;  j++; }
   else { //匹配失败，j返回子串首，i回退到下次匹配的起点
        i = i - j + 1;  j = 0;  }}
```



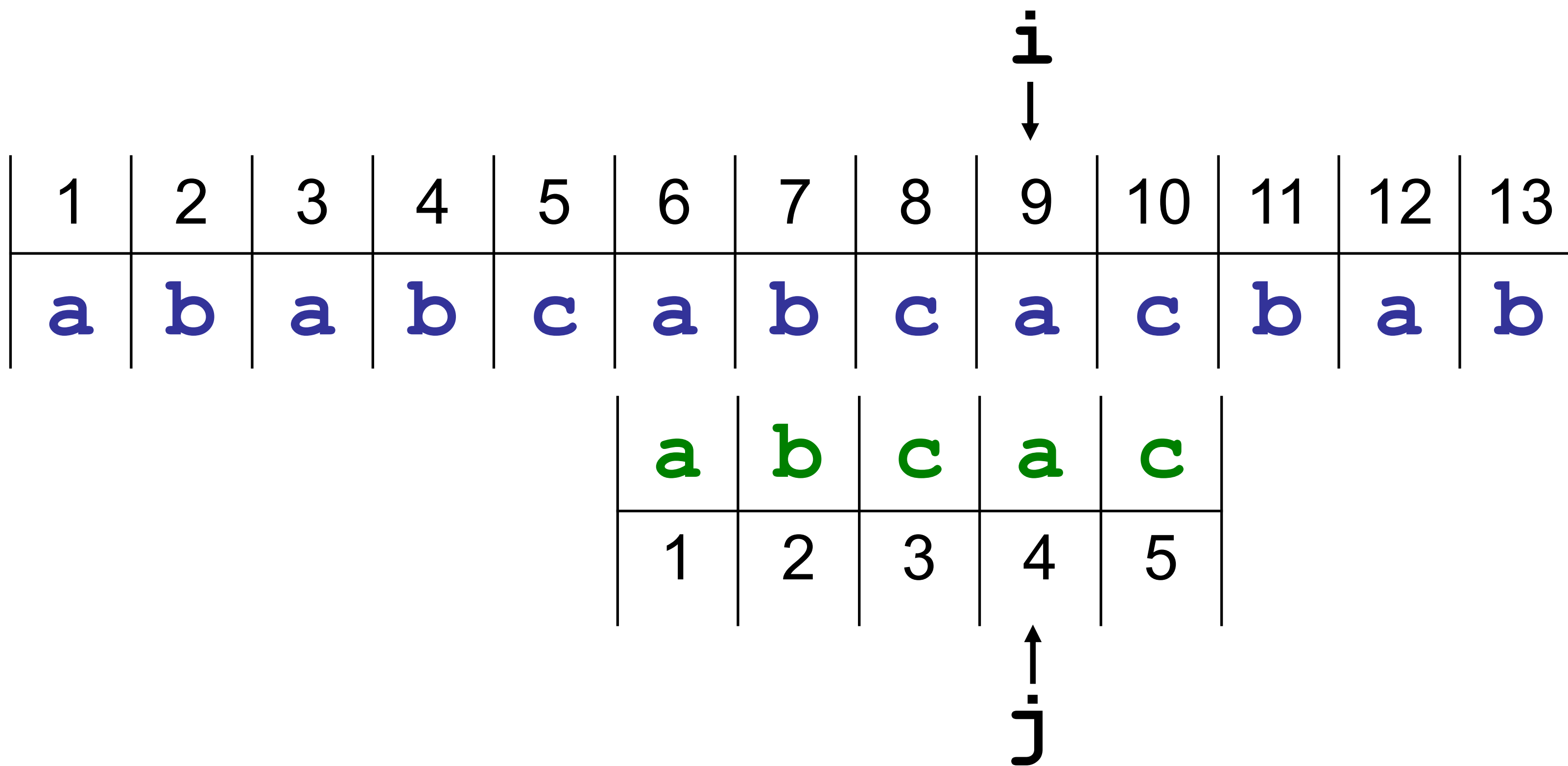
```
while(i < S._size && j < T._size ){
   if(S.s[i] == T.s[j]){ //当前字符匹配，i,j递增
        i++;  j++; }
   else { //匹配失败，j返回子串首，i回退到下次匹配的起点
        i = i - j + 1;  j = 0;  }}
```



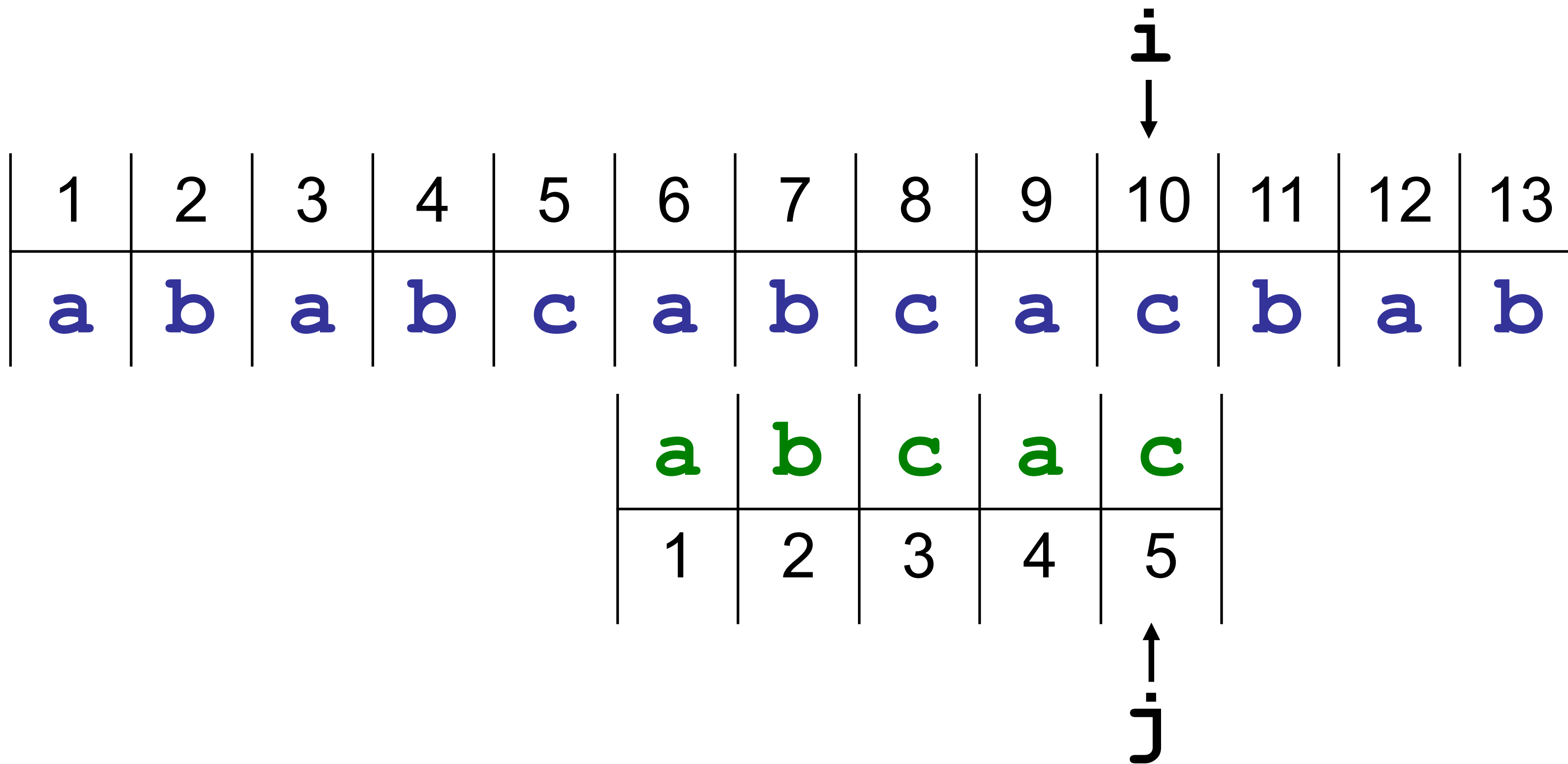
```
while(i < S._size && j < T._size ){
   if(S.s[i] == T.s[j]){ //当前字符匹配，i,j递增
        i++;  j++; }
   else { //匹配失败，j返回子串首，i回退到下次匹配的起点
        i = i - j + 1;  j = 0;  }}
```



```
while(i < S._size && j < T._size ){
   if(S.s[i] == T.s[j]){ //当前字符匹配, i,j递增
        i++;  j++; }
   else { //匹配失败, j返回子串首, i回退到下次匹配的起点
        i = i - j + 1;  j = 0;  }}
```

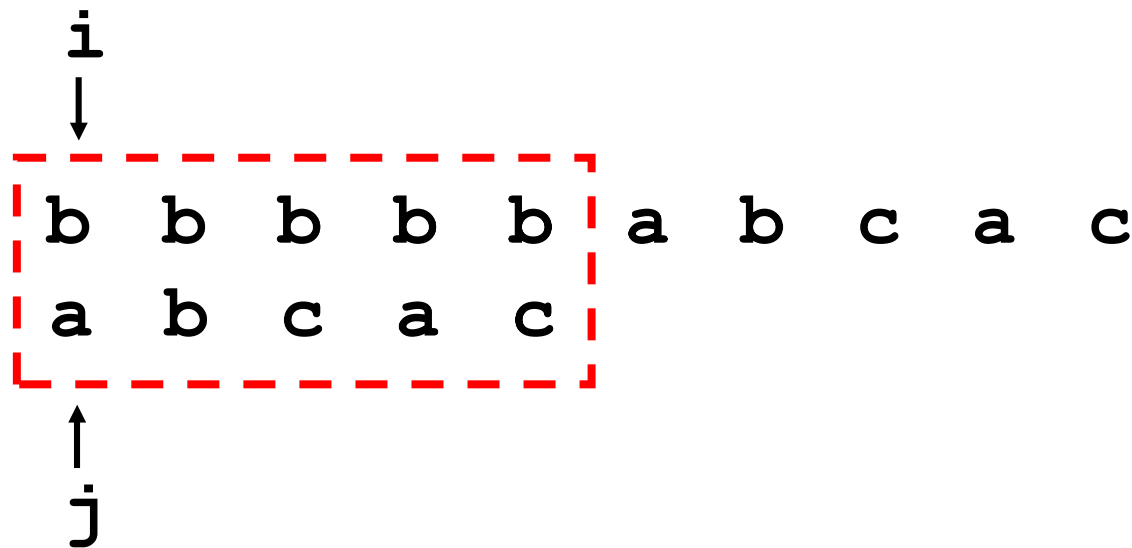


```
while(i < S._size && j < T._size ){
   if(S.s[i] == T.s[j]){ //当前字符匹配，i,j递增
       i++;  j++; }
   else { //匹配失败，j返回子串首，i回退到下次匹配的起点
       i = i - j + 1;  j = 0;  }}
```



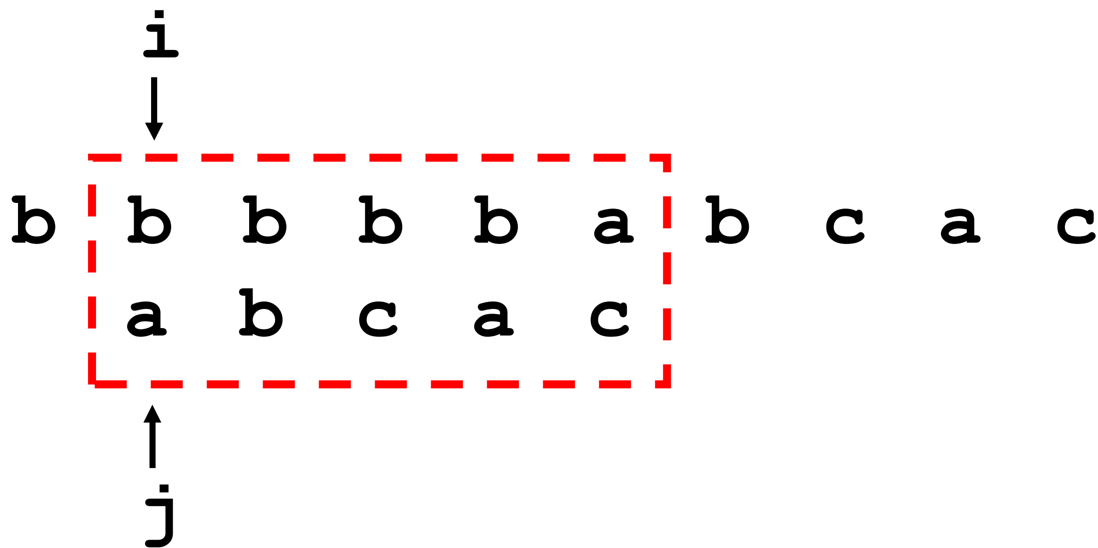
串的模式匹配：简单算法

- 算法分析
  - 最好的情况
    - 主串**S**和模式**T**中的每个字符都只访问了一次
    - 复杂度 = **O(n + m)**



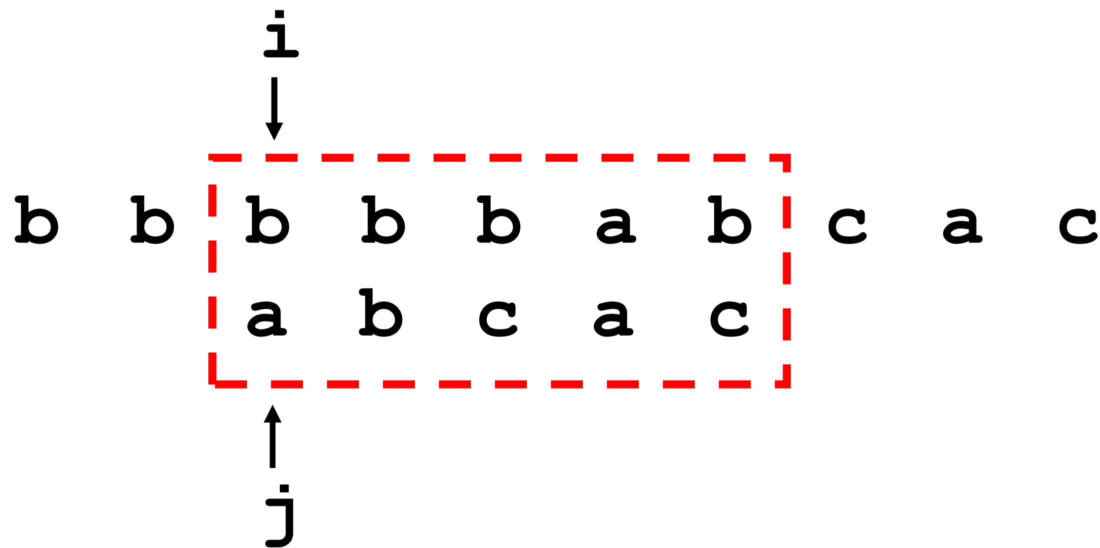
串的模式匹配：简单算法

- 算法分析
  - 最好的情况
    - 主串**S**和模式**T**中的每个字符都只访问了一次
    - 复杂度 = **O(n + m)**



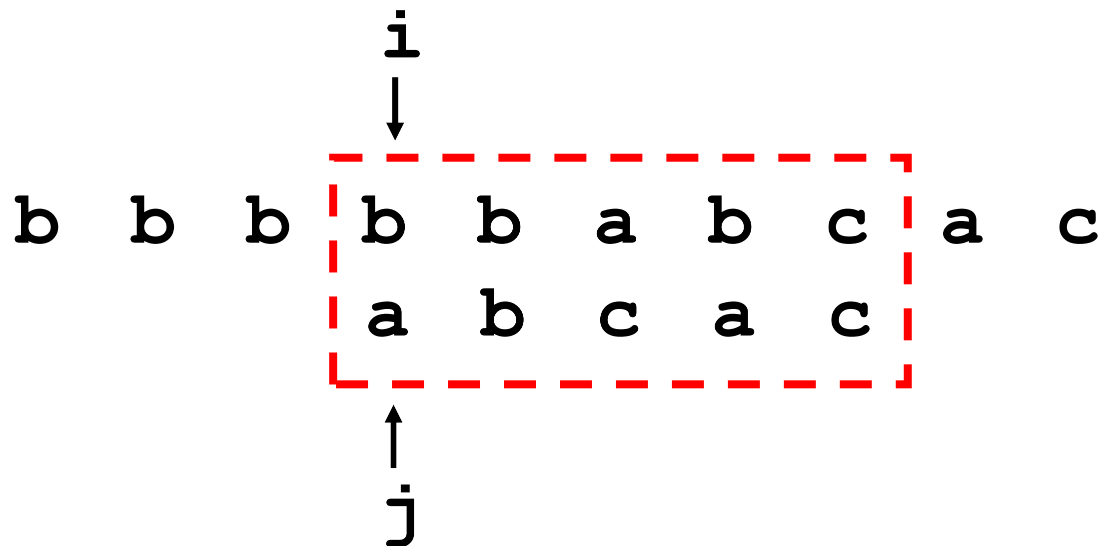
串的模式匹配：简单算法

- 算法分析
  - 最好的情况
    - 主串**S**和模式**T**中的每个字符都只访问了一次
    - 复杂度 = **O(n + m)**



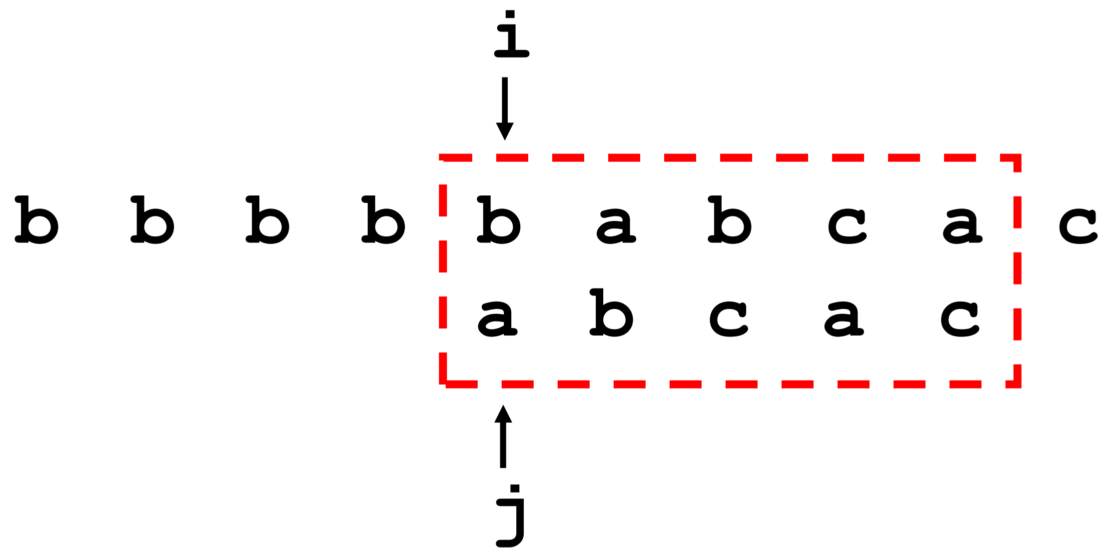
串的模式匹配：简单算法

- 算法分析
  - 最好的情况
    - 主串**S**和模式**T**中的每个字符都只访问了一次
    - 复杂度 = **O(n + m)**



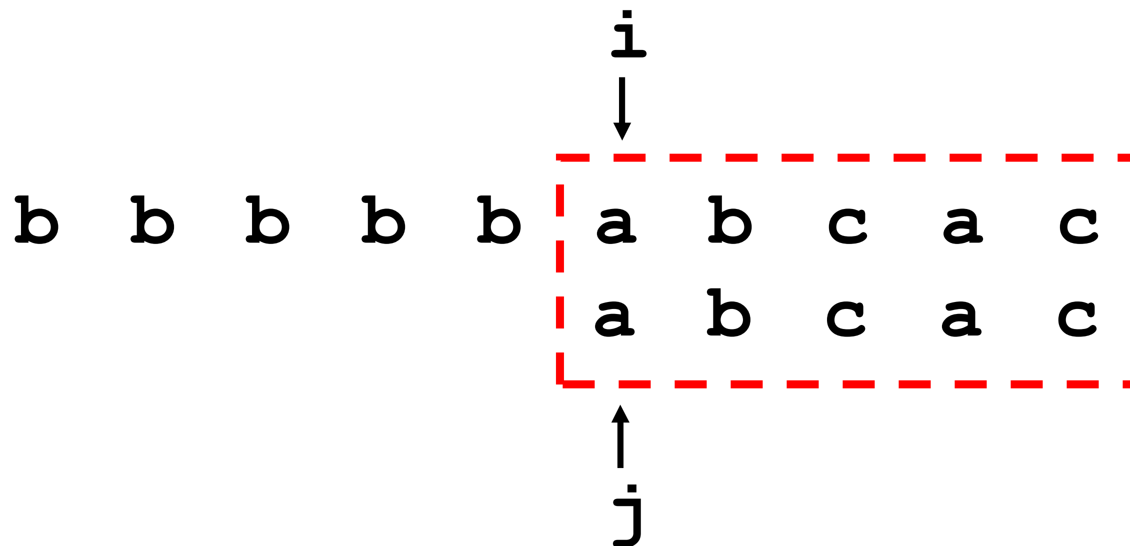
串的模式匹配：简单算法

- 算法分析
  - 最好的情况
    - 主串**S**和模式**T**中的每个字符都只访问了一次
    - 复杂度 = **O(n + m)**



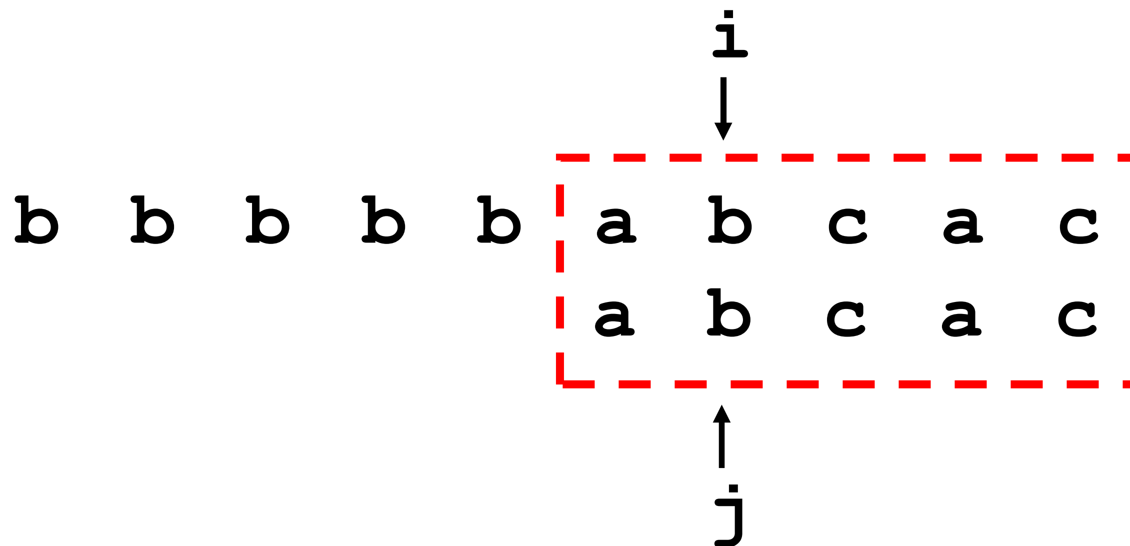
串的模式匹配：简单算法

- 算法分析
  - 最好的情况
    - 主串**S**和模式**T**中的每个字符都只访问了一次
    - 复杂度 = **O(n + m)**



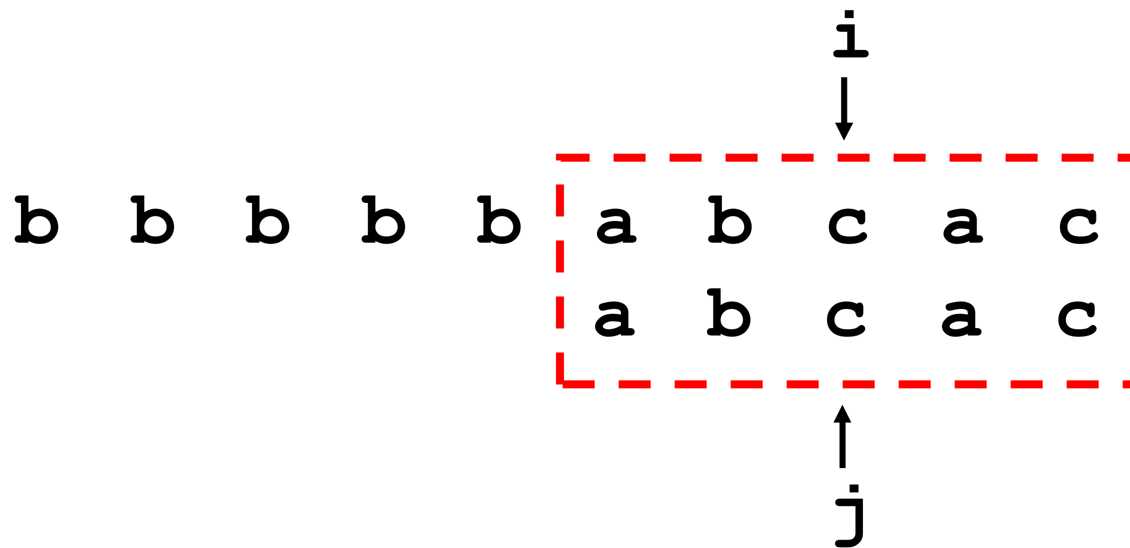
串的模式匹配：简单算法

- 算法分析
  - 最好的情况
    - 主串**S**和模式**T**中的每个字符都只访问了一次
    - 复杂度 = O(n + m)



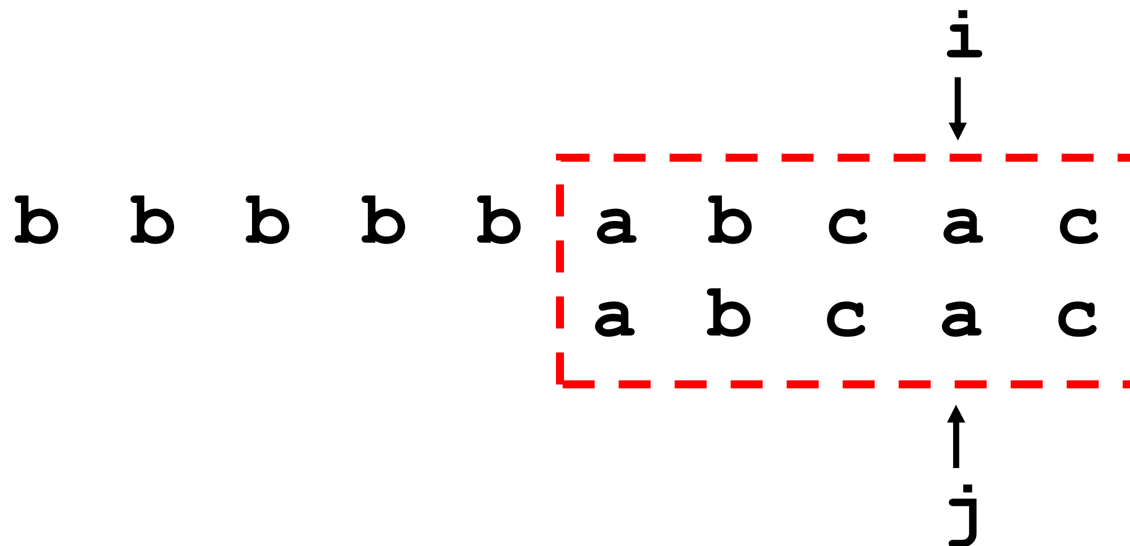
串的模式匹配：简单算法

- 算法分析
  - 最好的情况
    - 主串**S**和模式**T**中的每个字符都只访问了一次
    - 复杂度 = **O(n + m)**



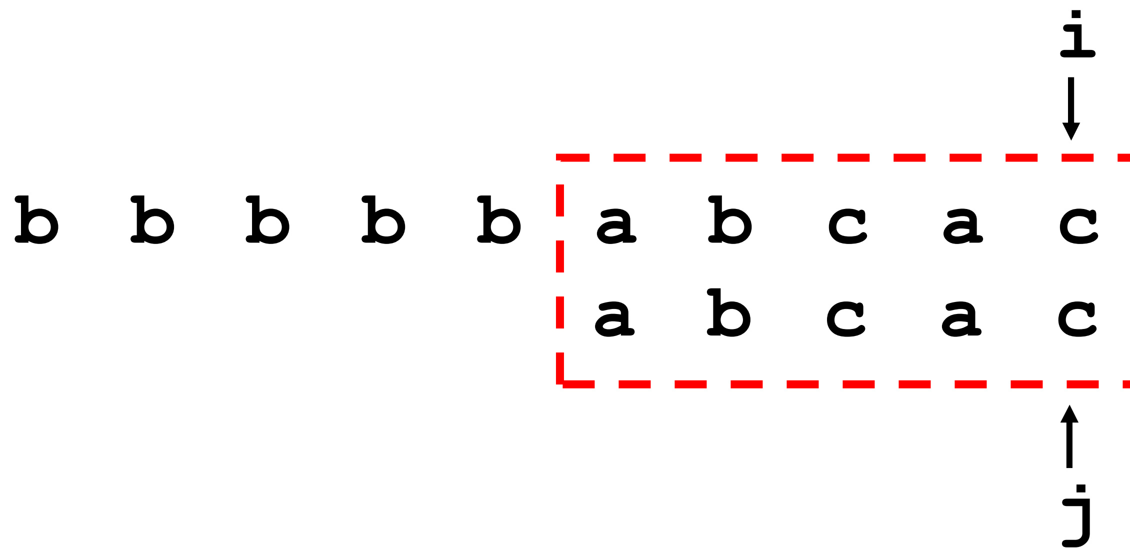
串的模式匹配：简单算法

- 算法分析
  - 最好的情况
    - 主串**S**和模式**T**中的每个字符都只访问了一次
    - 复杂度 = **O(n + m)**



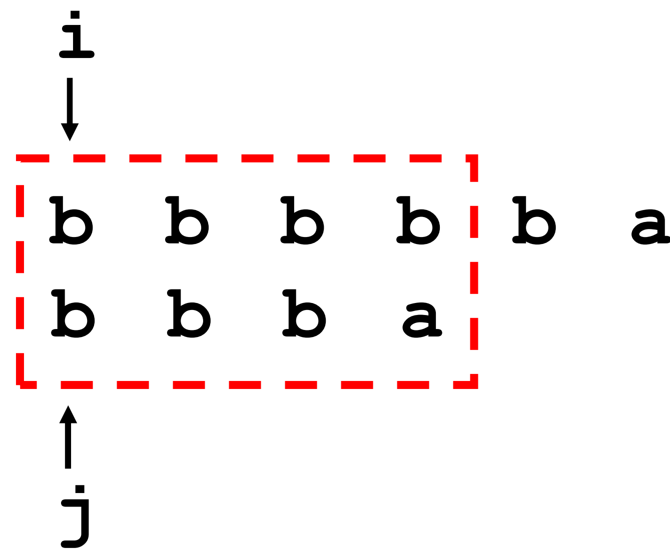
串的模式匹配：简单算法

- 算法分析
  - 最好的情况
    - 主串**S**和模式**T**中的每个字符都只访问了一次
    - 复杂度 = **O(n + m)**



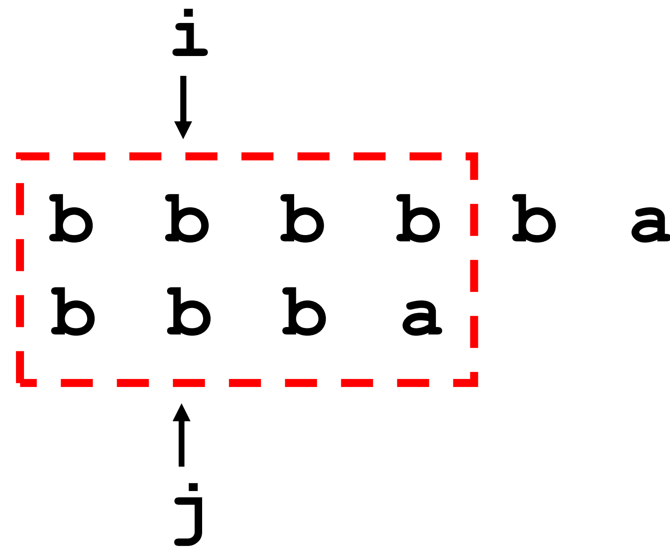
串的模式匹配：简单算法

- 最差的情况
  - 主串**S**中的每个字符，分别和模式**T**中的每个字符匹配一次
  - 复杂度 **= O(n * m)**



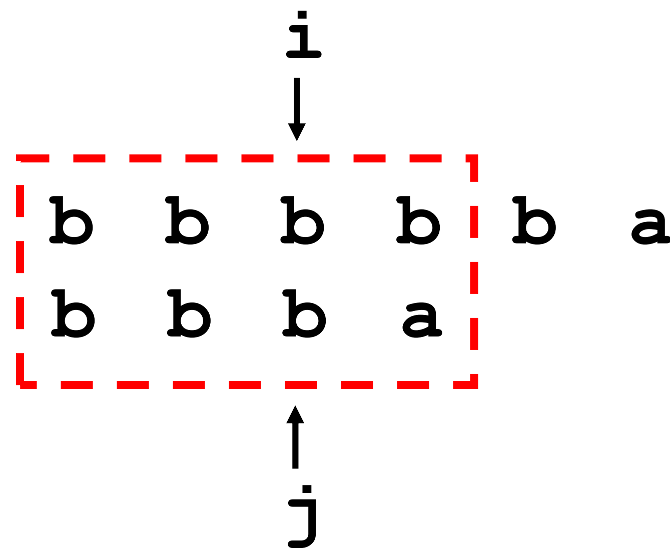
串的模式匹配：简单算法

- 最差的情况
  - 主串**S**中的每个字符，分别和模式**T**中的每个字符匹配一次
  - 复杂度 **= O(n * m)**



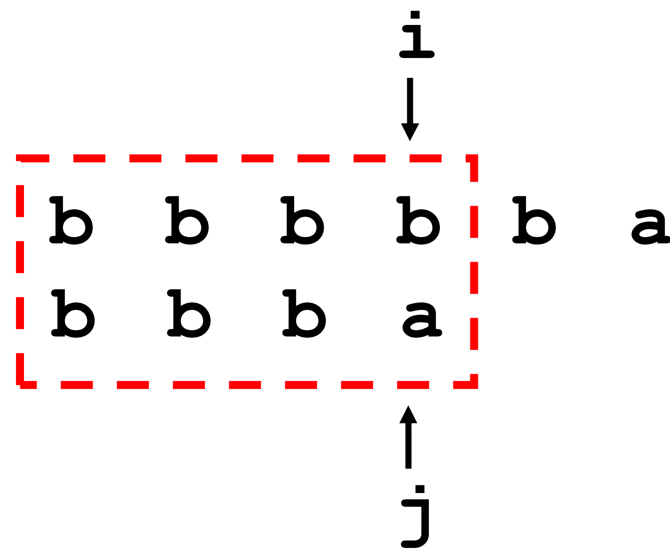
串的模式匹配：简单算法

- 最差的情况
  - 主串**S**中的每个字符，分别和模式**T**中的每个字符匹配一次
  - 复杂度 = **O(n * m)**



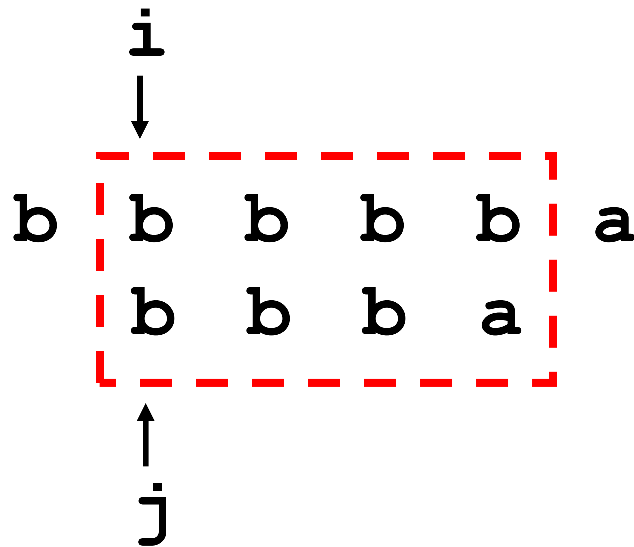
串的模式匹配：简单算法

- 最差的情况
  - 主串S中的每个字符，分别和模式T中的每个字符匹配一次
  - 复杂度 = O(n * m)



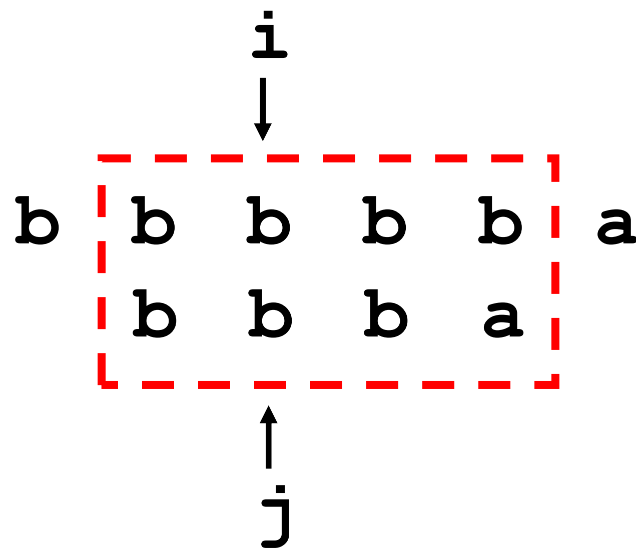
串的模式匹配：简单算法

– 最差的情况
  - 主串**S**中的每个字符，分别和模式**T**中的每个字符匹配一次
  - 复杂度 = **O(n * m)**



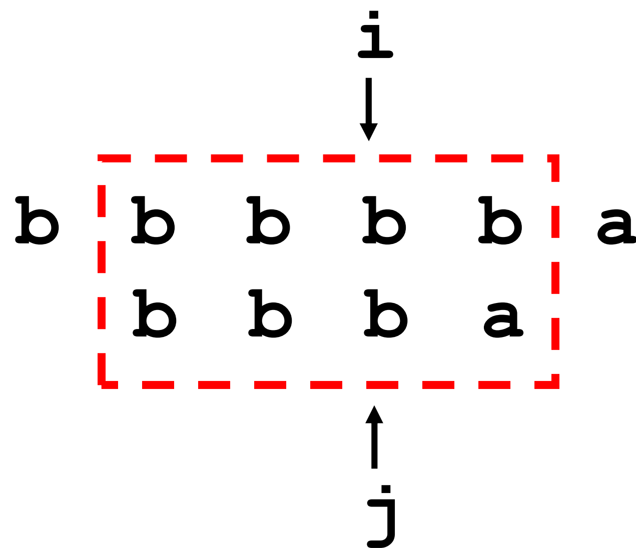
串的模式匹配：简单算法

- 最差的情况
  - 主串**S**中的每个字符，分别和模式**T**中的每个字符匹配一次
  - 复杂度 **= O(n * m)**



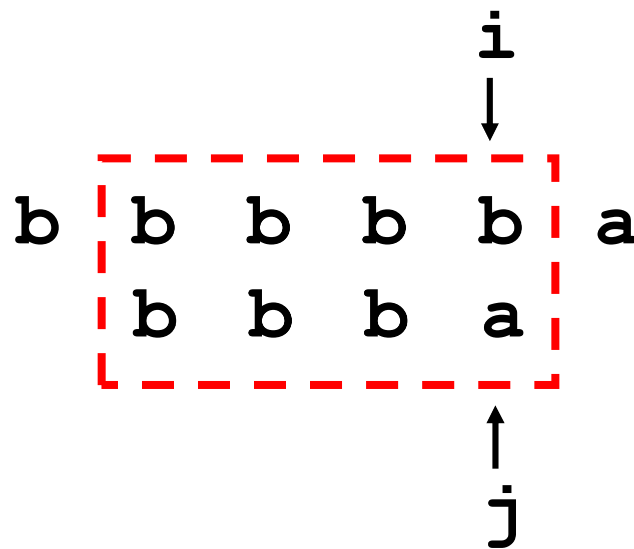
串的模式匹配：简单算法

- 最差的情况
  - 主串**S**中的每个字符，分别和模式**T**中的每个字符匹配一次
  - 复杂度 **= O(n * m)**



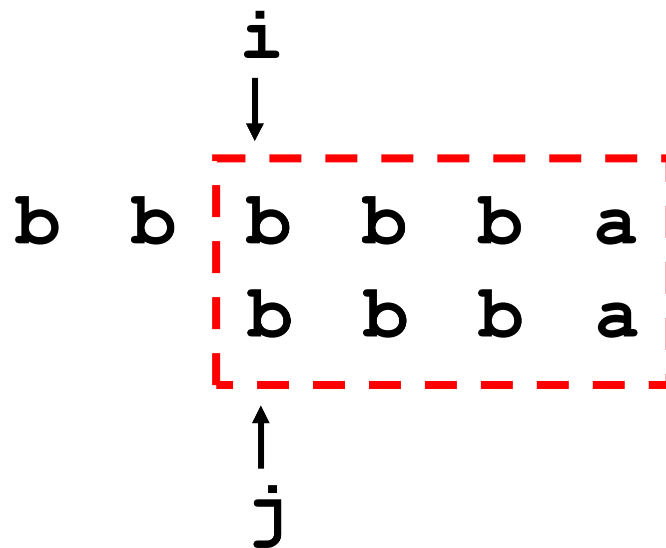
串的模式匹配：简单算法

- 最差的情况
  - 主串**S**中的每个字符，分别和模式**T**中的每个字符匹配一次
  - 复杂度 = **O(n * m)**



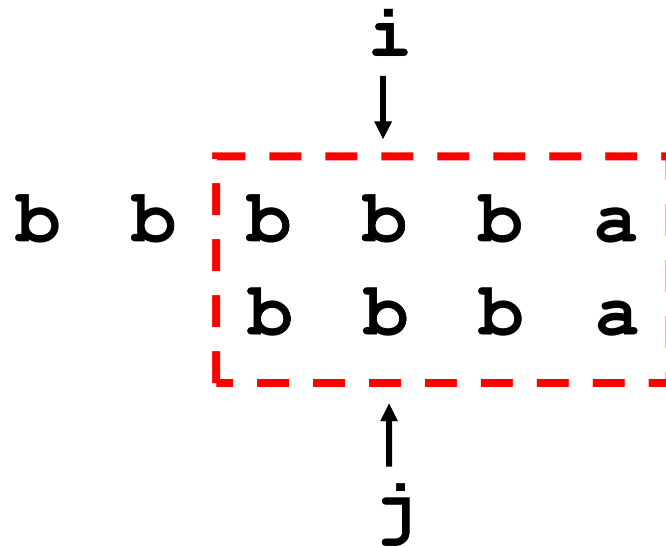
串的模式匹配：简单算法

- 最差的情况
  - 主串**S**中的每个字符，分别和模式**T**中的每个字符匹配一次
  - 复杂度 = **O(n * m)**



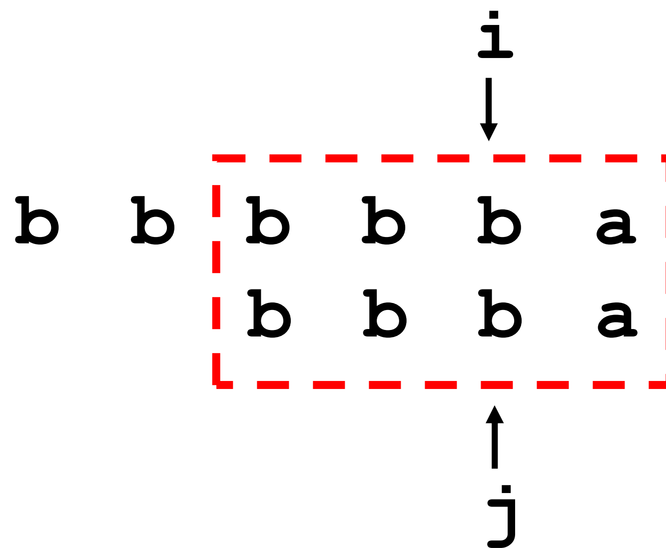
串的模式匹配：简单算法

- 最差的情况
  - 主串**S**中的每个字符，分别和模式**T**中的每个字符匹配一次
  - 复杂度 = **O(n * m)**



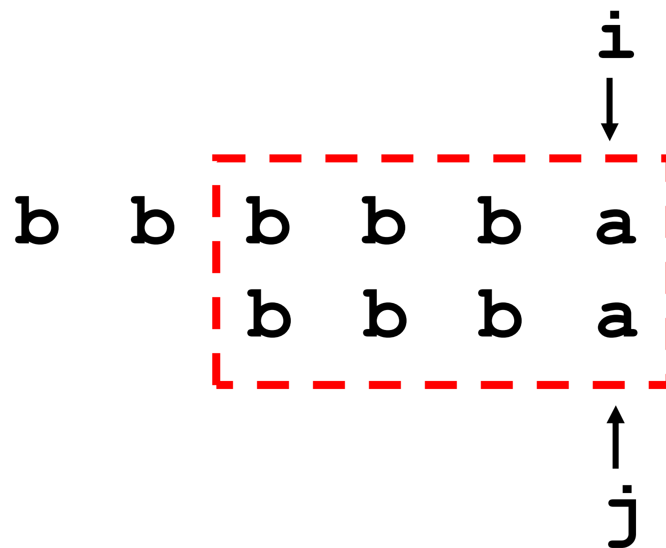
串的模式匹配：简单算法

- 最差的情况
  - 主串**S**中的每个字符，分别和模式**T**中的每个字符匹配一次
  - 复杂度 = **O(n * m)**



串的模式匹配：简单算法

- 最差的情况
  - 主串S中的每个字符，分别和模式T中的每个字符匹配一次
  - 复杂度 = O(n * m)



串的模式匹配：KMP算法

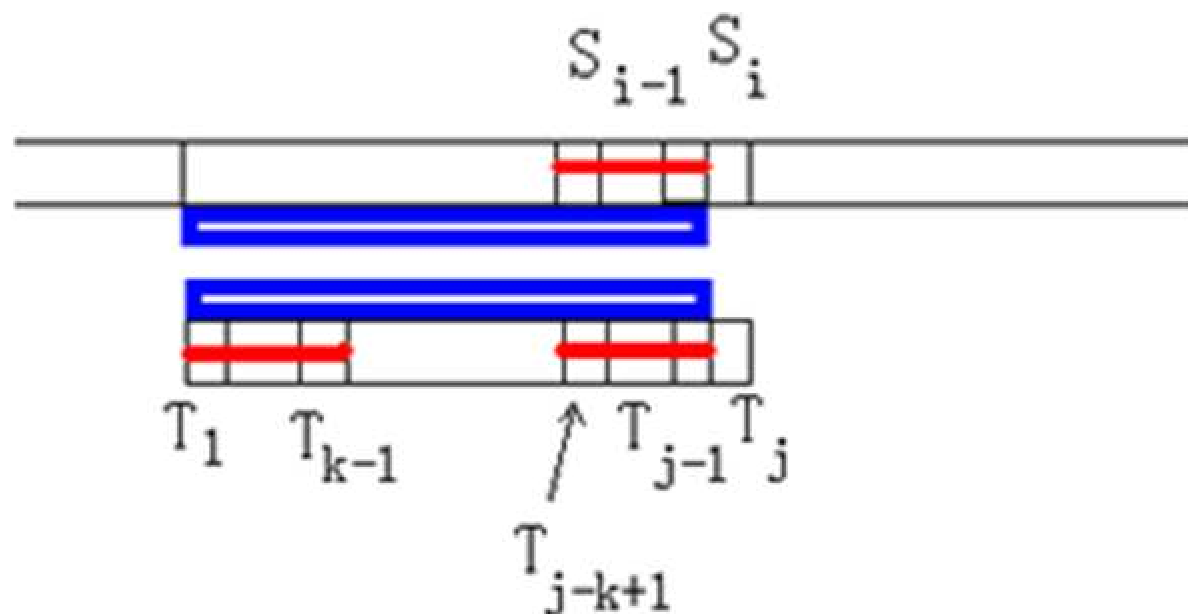
- **KMP算法**
  - 时间复杂度可以达到**O(m+n)**
  - 基本思想：在简单算法的基础上
    - **i**不要回退
    - 模式串尽量多往右移

串的模式匹配：KMP算法

【问题一】

在模式匹配过程中，若要保证主串指针 i 不回溯，则当主串的第 i 个字符与模式串的第 j 个字符失配时，下一次的比较应在哪两个字符间进行？

【分析】

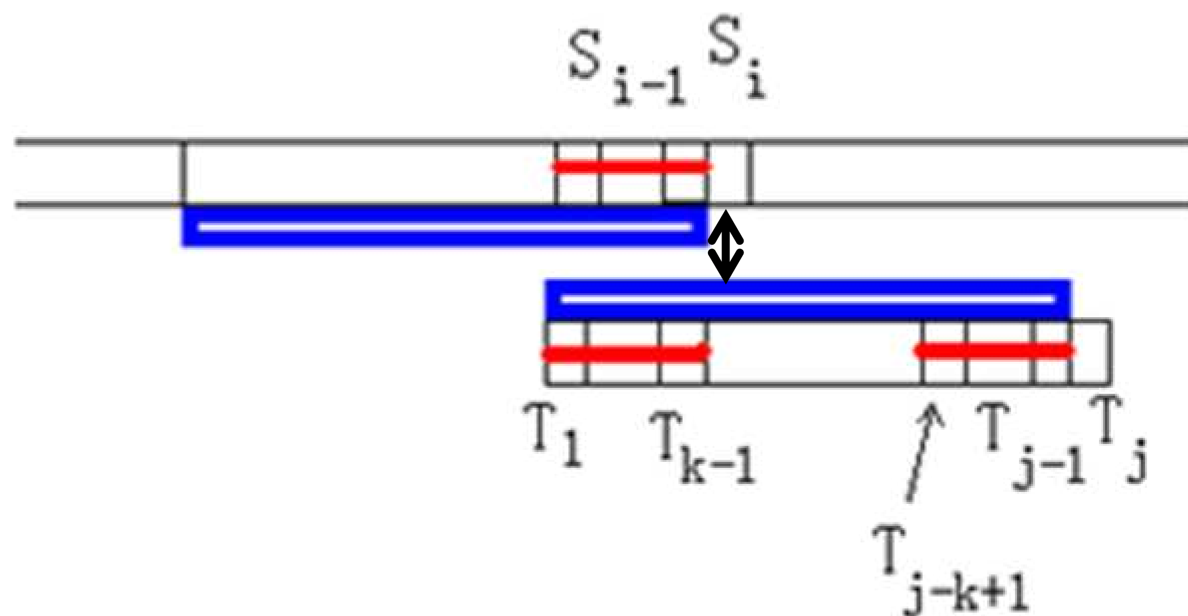


串的模式匹配：KMP算法

【问题一】

在模式匹配过程中，若要保证主串指针 i 不回溯，则当主串的第 i 个字符与模式串的第 j 个字符失配时，下一次的比较应在哪两个字符间进行？

【分析】



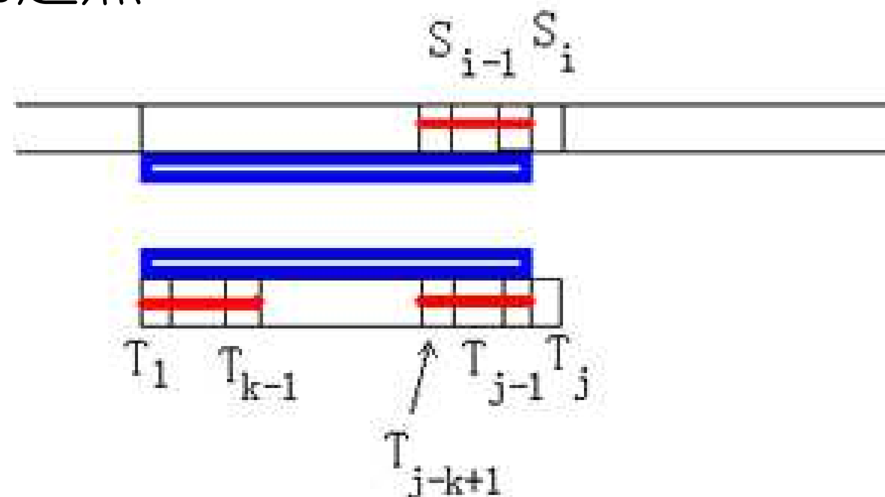
串的模式匹配：KMP算法

若存在最大的k满足‘$t_{j-k+1}\ldots t_{j-1}$’=‘$t_1\ldots t_{k-1}$’，则可以将模式串向右滑行k-1个，即下一步从t_k开始和主串的S_i比较。为什么？

证明（反证法）：

如果存在**l>k**,使下一步可从**tl**开始和主串的**Si**比较.则**l**也满足‘**tj-l+1…tj-1**’=‘**t1…tl-1**’，从而矛盾。

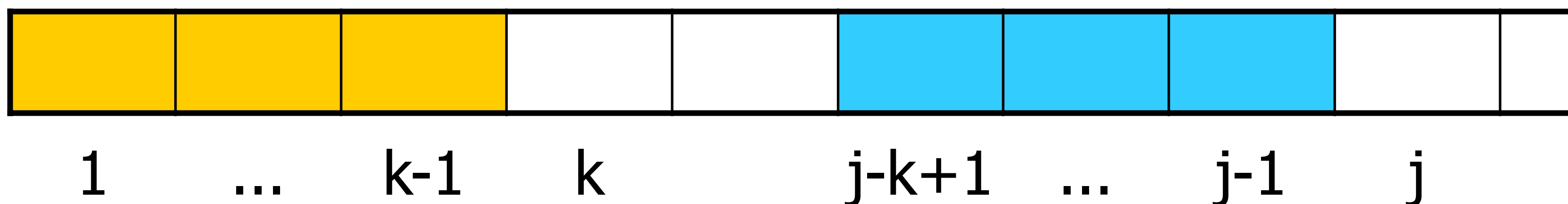
同样,如果从**l<k**进行比较,则可能滑行过远,错过了可以匹配的. 因此**k**是下次模式最好的起点.



串的模式匹配：KMP算法

- 模式串的**next**函数

$$next[j]=\begin{cases}0 & j=1\\ Max\{k|1<k<j \text{ 且 } 'p_1\cdots p_{k-1}'='p_{j-k+1}\cdots p_{j-1}'\} \\ 1 & others\end{cases}$$



  - 意义：**j**之前的子串中，左起一段**=**右起一段，最长不超过**k**

串的模式匹配：KMP算法

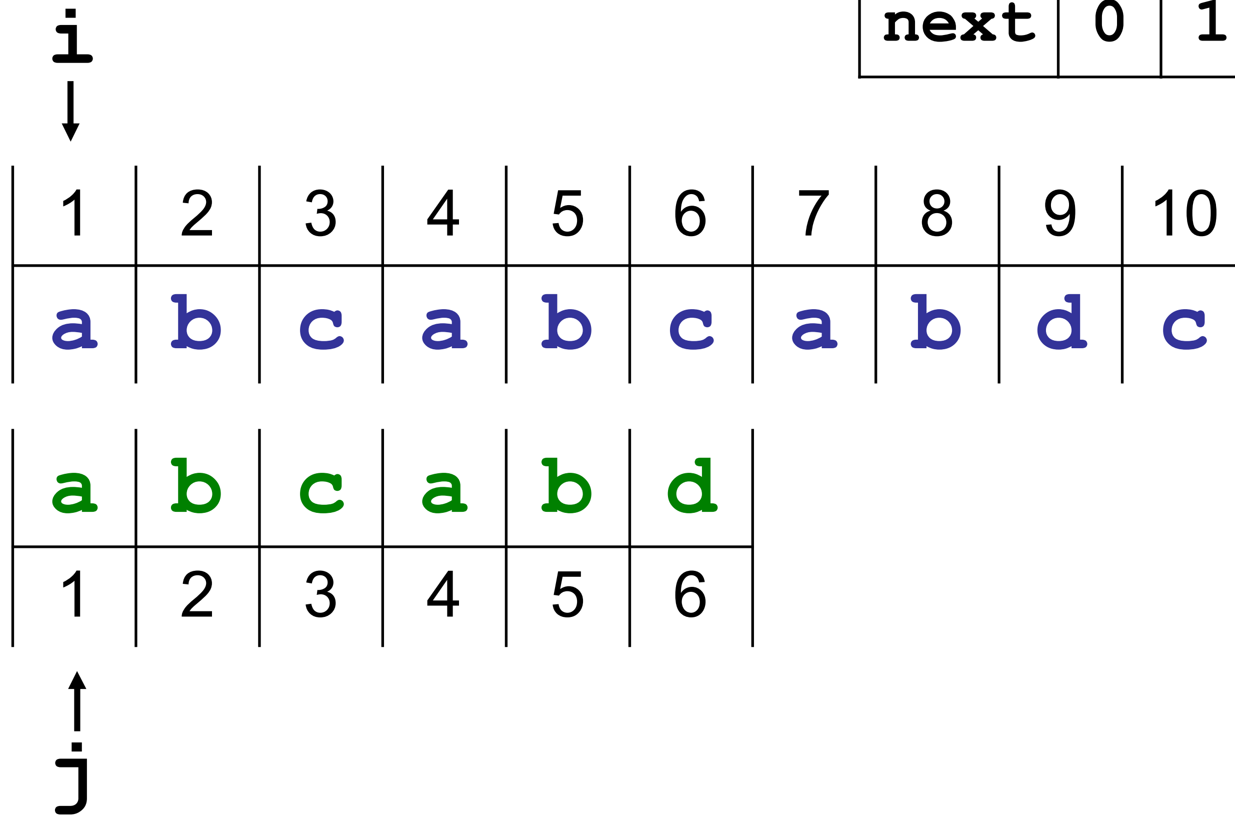
– 例如：

j	1	2	3	4	5	6	7	8
模式串	a	b	a	a	b	c	a	c
next[j]	0	1	1	2	2	3	1	2

$$next[j]=\begin{cases} 0 & j=1 \\ \mathrm{Max}\{k \mid 1<k<j \text{ 且 } 'p_1 \cdots p_{k-1}' = 'p_{j-k+1} \cdots p_{j-1}'\} \\ 1 & \text{others} \end{cases}$$

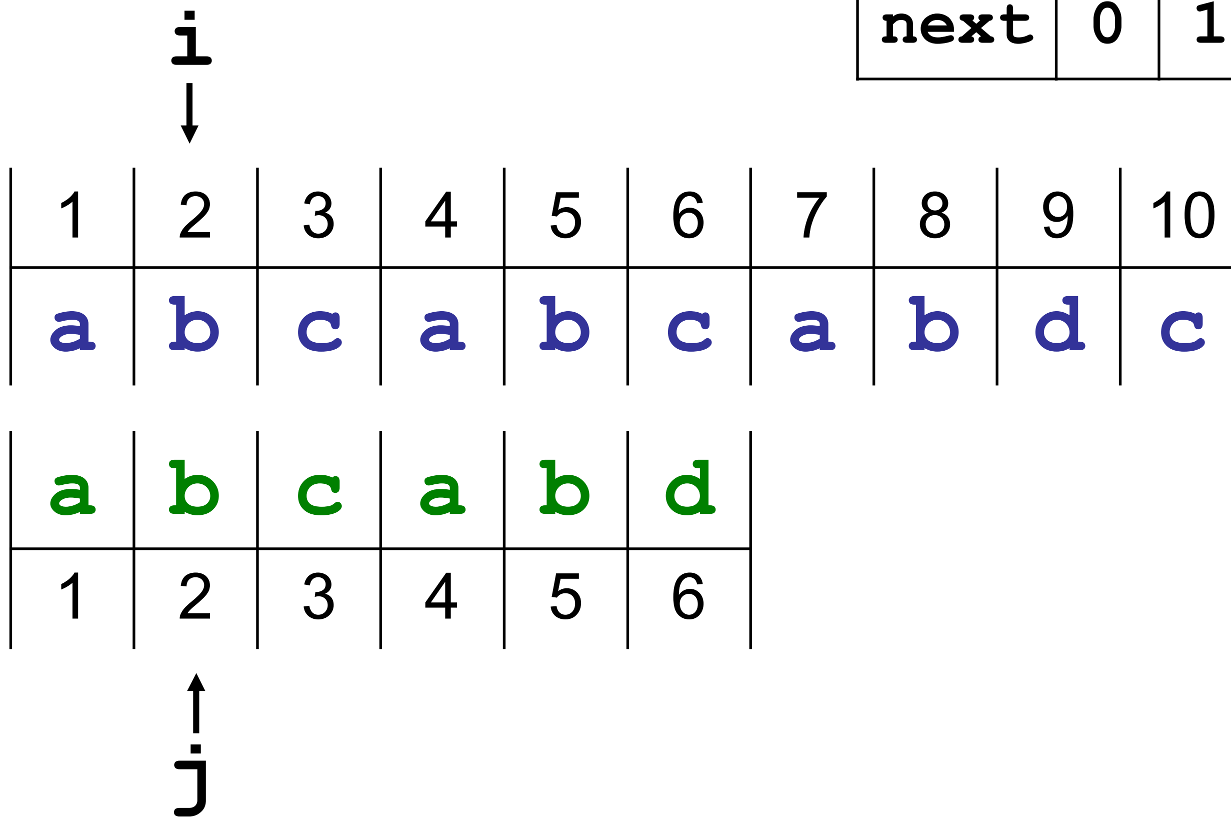
串的模式匹配：KMP算法

	1	2	3	4	5	6
子串	a	b	c	a	b	d
next	0	1	1	1	2	3



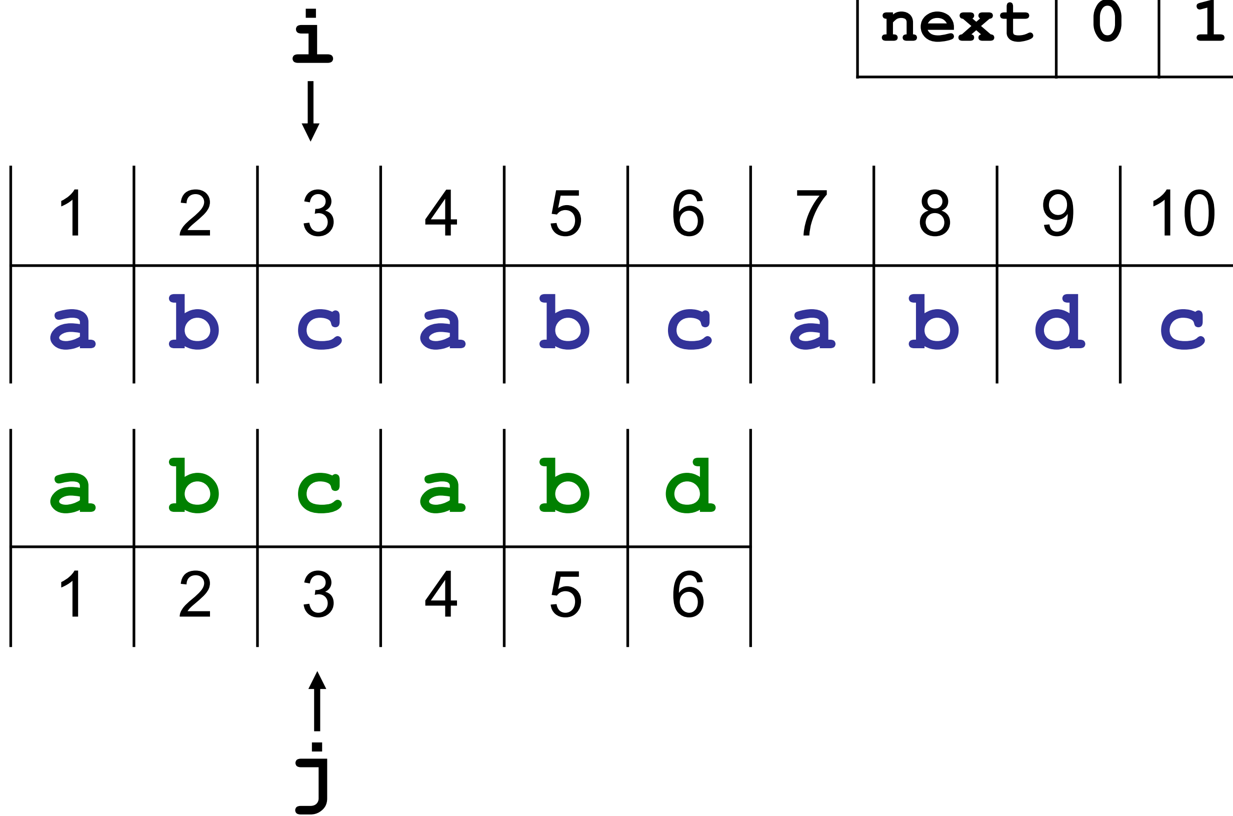
串的模式匹配：KMP算法

	1	2	3	4	5	6
子串	a	b	c	a	b	d
next	0	1	1	1	2	3



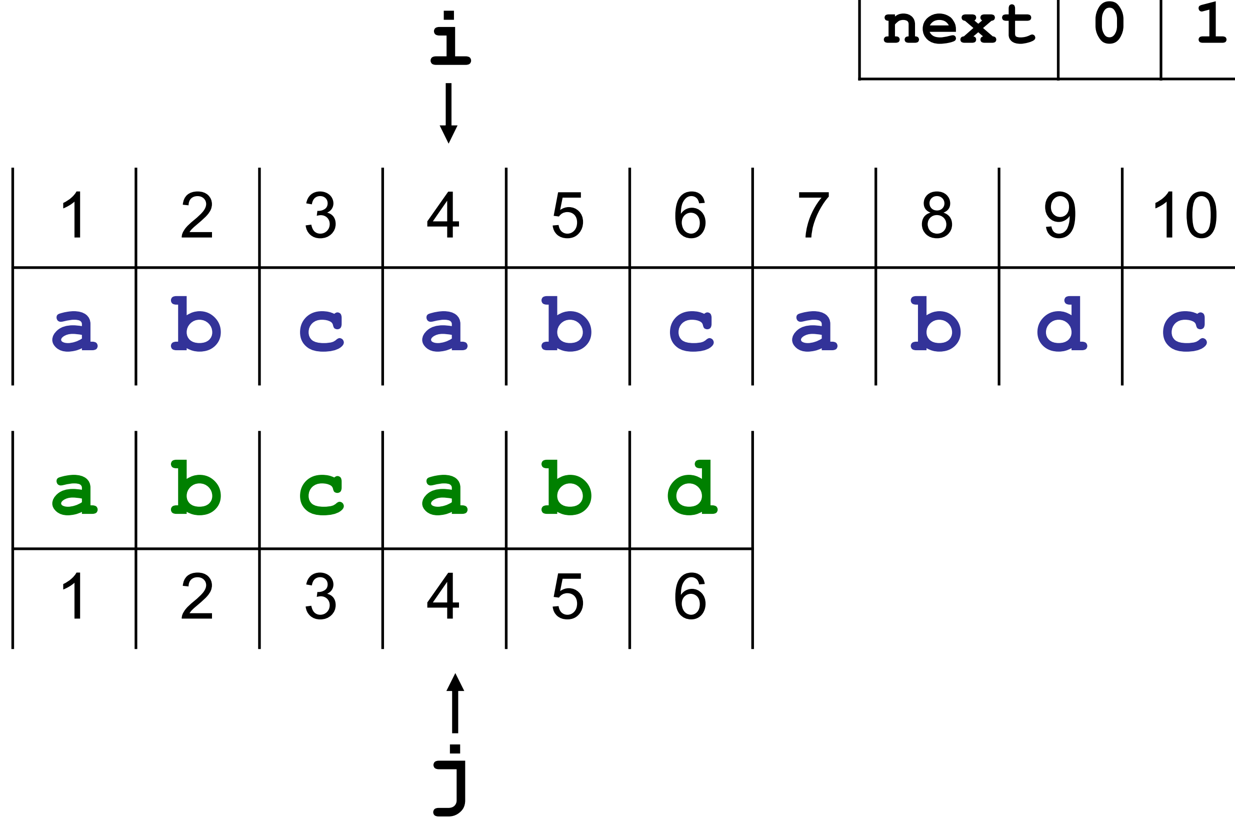
串的模式匹配：KMP算法

	1	2	3	4	5	6
子串	a	b	c	a	b	d
next	0	1	1	1	2	3



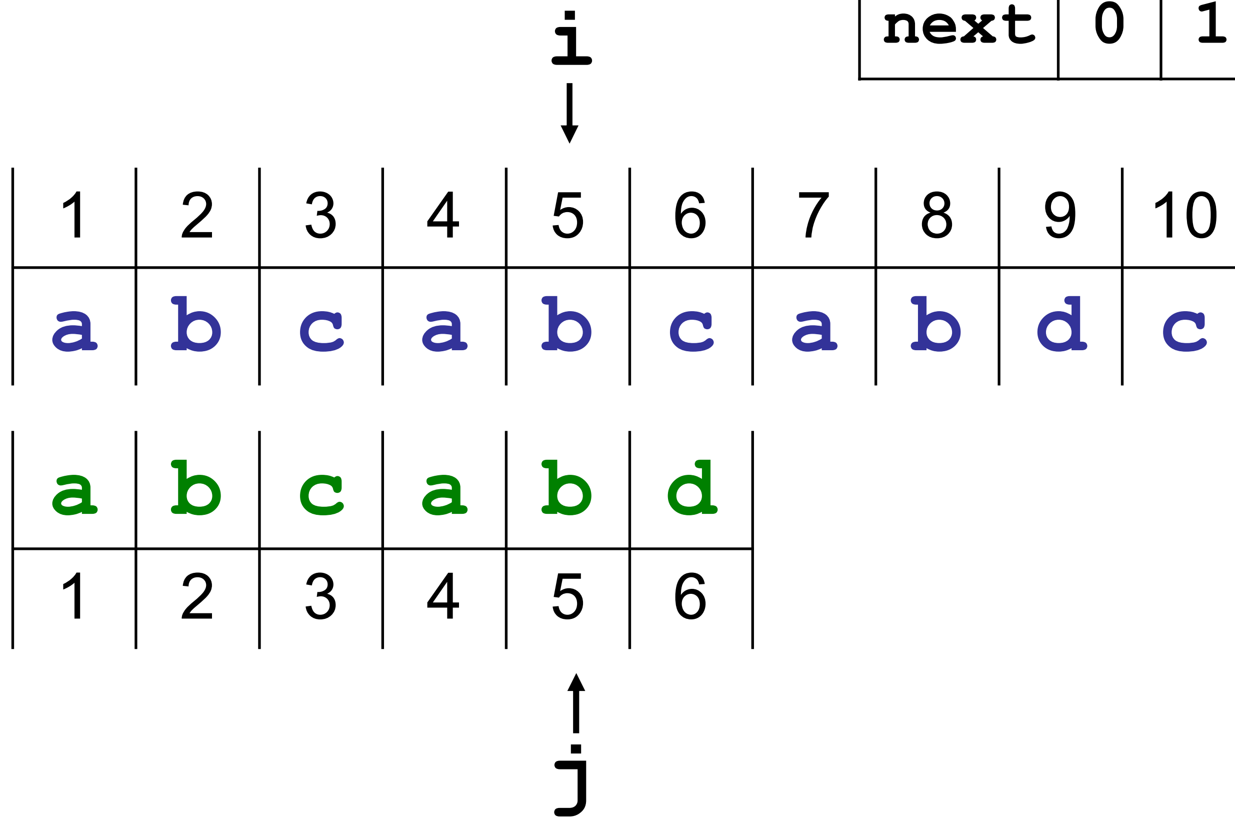
串的模式匹配：KMP算法

	1	2	3	4	5	6
子串	a	b	c	a	b	d
next	0	1	1	1	2	3



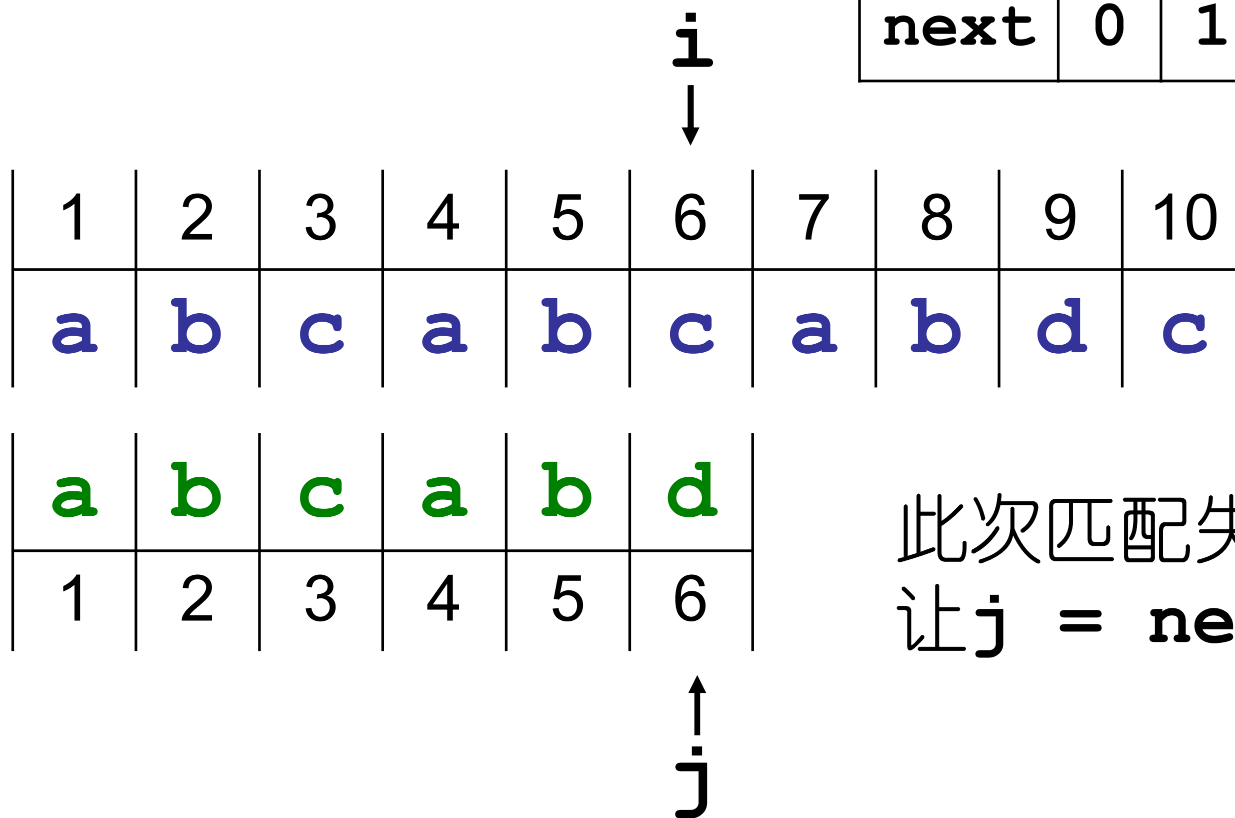
串的模式匹配：KMP算法

	1	2	3	4	5	6
子串	a	b	c	a	b	d
next	0	1	1	1	2	3



串的模式匹配：KMP算法

	1	2	3	4	5	6
子串	a	b	c	a	b	d
next	0	1	1	1	2	3



此次匹配失败

让j = next[j] = 3

KMP算法:next函数计算

【问题二】

设给定模式串 T，求其对应的**Next(j)**函数。

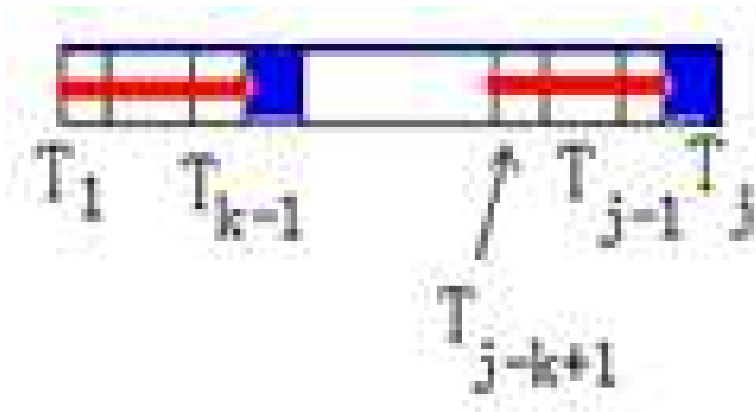
$$next[j]=\begin{cases}0 & j=1\text{时}\\ Max\{k \mid 1<k<j\text{且}'t_1\ldots t_{k-1}'='t_{j-k+1}\ldots t_{j-1}'\} & \text{该集合不空时}\\ 1 & \text{其它情况}\end{cases}$$

KMP算法:next函数计算

1)next[1]=0

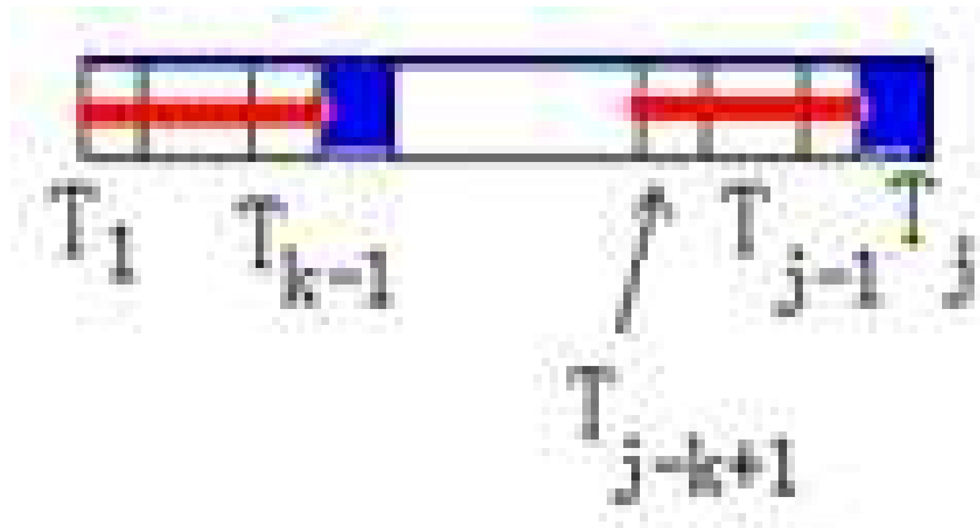
2)设 next[j]=k,求next[j+1]:

若tk = tj, 则next[j+1]=k+1.



KMP算法:next函数计算

若tj≠tk，则相当于以图中T为主串，T为子串进行匹配时，主串中第 j 个字符与子串中第 k 个字符失配，此时可令k=next[k]，再对tj和tk进行比较，如此循环，直至tj=tk 或k=0为止，此时next[j+1]=k+1。



KMP算法:next函数计算

- ***next*函数的计算**
  - 一个递归的过程:
  - 已知 **next[1]=0**
  - 若 **next[j]=k**
    - 说明有 ‘$t_1...t_{k-1}$’=‘$t_{j-k+1}...t_{j-1}$’
    - 若$t_k=t_j$，则**next[j+1]=k+1**
    - 若$t_k!=t_j$，令**k’=next[k]**
      - 若$t_{k'}=t_j$，则**next[j+1]=k’+1**
      - 若$t_{k'}!=t_j$，则尝试**next[k’]...**

KMP算法:next函数计算

- 示例
  - 首先next[0]=1
  - 假设已知next[j]=k
  - next[j+1]=?

	1	2	3	4	5	6	7	8	9	10	11	12	13
子串	a	b	c	a	b	d	a	b	c	a	b	c	a
next	0											6	?

k (指向 6)　j (指向 12)

KMP算法:next函数计算

- 示例
  - 若T[k]=T[j]
  - 则next[j+1]=k+1

	1	2	3	4	5	6 (k)	7	8	9	10	11	12 (j)	13
子串	a	b	c	a	b	d	a	b	c	a	b	d	a
next	0											6	7

KMP算法:next函数计算

- 示例
  - 若T[k]!=T[j]
  - 令k′=next[k]
  - 若T[k′]=T[j]，则next[j+1]=k′+1

	1	2	3	4	5	6	7	8	9	10	11	12	13
子串	a	b	c	a	b	d	a	b	c	a	b	c	a
next	0											6	4

k′ → 3, k → 6, j → 12

KMP算法:next函数计算

- 为什么令k′=next[k]？
  - next[12]=6，说明T[1..5]=T[7..11]
  - k′=next[k]=3，说明T[1..2]=T[4..5]
  - 则T[1..2]=T[10..11]
  - 若又有T[k′]=T[j]，则next[j+1]=k′+1

	1	2	3 (k′)	4	5	6 (k)	7	8	9	10	11	12 (j)	13
子串	a	b	c	a	b	d	a	b	c	a	b	c	a
next	0											6	4

KMP算法:next函数计算

- 计算**next**的函数

```
void get_next(String T, int &next[])
{
  j = 0; next[0] = 0; k = 0;
  while(j < T._size-1) {
     if( k==0 || T.s[j] == T.s[k]) {
          j ++;  k ++;  next[j] = k;
     }
     else      k = next[k];
  }
}
```

KMP算法：

【问题三】

设已求得模式串对应的**next**数组中各元素的值，设计模式匹配算法。

【分析】

(1)令 i 指向 S 中第 pos 个字符，j 指向 T 中第 1 个字符；

(2)将 S 中当前字符与 T 中当前字符进行比较，
若相等，则令i++，j++；
若不等，则i不变，令j=next[j]，若得j值为0，则应使主串中下一字符与模式串中第1个字符进行比较，也为i++，j++；

(3)反复执行(2)直至模式匹配完成。

KMP算法:next函数计算

- 匹配函数

```
int find_KMP(String S, String T, int pos)
{
  i = pos-1;    j = 0;
  while(i < S._size && j < T._size) {
     if(j == 0 || S.s[i] == T.s[j]){
          i ++;      j ++; }
     else j = next[j];          //模式串右移
  }
  if(j >= T._size)   return i-j+1; //匹配成功
  else            return 0;         //失败
}
```

本章小结

- 串的类型定义
- 串的表示和实现
  - 单个字符数组表示
  - 结构表示法
  - 块链存储
- 串的模式匹配
  - 简单算法
  - **KMP**算法