《算法设计与分析》

第八章 蛮力法

马丙鹏

2023年11月27日

中国科学院大学
University of Chinese Academy of Sciences

第八章 蛮力法

- 8.1 概述
- 8.2 串匹配问题

8.1 概述

■ 1. 设计思想

- □穷举法，也称枚举法，是一种简单直接地解决问题的方法，采用一定的策略依次处理待求解问题的所有元素，从而找出问题的解。
- □最容易应用的方法
- □穷举法设计的算法其时间性能往往也是最低的，典型的指数时间算法一般都是通过蛮力穷举得到的。
- □穷举法的关键：依次处理所有元素

① 确定穷举的范围

② 保证处理过的元素不再被处理（为了避免陷入重复试探）

8.1 概述

- **1. 设计思想**
  - 用穷举法解决问题，通常可以从两个方面进行算法设计：

① 找出枚举范围：分析问题所涉及的各种情况。

② 找出约束条件：分析问题的解需要满足的条件，并用逻辑表达式表示

8.1 概述

■ 1. 设计思想

- □例： 求所有的三位数，除以 11 所得的余数等于三个数字的平方和。

  ① 枚举范围：100 ~ 999，共 900个。

  ② 约束条件：设三位数是n，其百位、十位、个位的数字分别为 x，y，z。则：

  $$n \% 11 == x^2 + y^2 + z^2$$

  ➢注意到 $x^2 + y^2 + z^2 \le 10$，可以缩小枚举范围：

  $$1 \le x \le 3,\quad 0 \le y \le 3,\quad 0 \le z \le 3$$

100,101

8.1 概述

■ 1. 设计思想

□基于以下原因，穷举法也是一种重要的算法设计技术：

① 穷举法不是一个最好的算法（巧妙和高效的算法很少出自蛮力），但当想不出更好的办法时，也是一种有效的解决问题的方法。

② 理论上，穷举法可以解决可计算领域的各种问题。对于一些基本的问题，穷举法是一种常用的算法设计技术。

③ 穷举法经常用来解决一些较小规模的问题。

④ 对于一些重要的问题，穷举法可以设计一些合理的算法，这些算法具有实用价值，而且不受问题规模的限制。

⑤ 穷举法可以作为某类问题时间性能的下界，来衡量同样问题的其他算法是否具有更高的效率。

8.1 概述

- 1. 设计思想
  - 蛮力法的一般格式：
    - 在蛮力法设计算法中，主要使用循环语句和选择语句
      - 循环语句用于穷举所有可能的情况，
      - 选择语句判定当前的条件是否为所求的解。

```
void Exhaustive(x, n, y, m)
{   for (int i=1; i<=n; i++)            //枚举x的所有可能的值
      for (int j=1; j<=m; j++)          //枚举y的所有可能的值
      {   …
          if (p(x[i], y[j]))
             输出一个解;
          …
      }
}
```

- x和y所有可能的搜索范围是笛卡尔积即 $([x_1, y_1], [x_1, y_2], \ldots, [x_1, y_m], \ldots, [x_n, y_1], [x_n, y_2], \ldots, [x_n, y_m])$。
- 这样的搜索范围可以用一棵树表示，称为解空间树。

8.1 概述

■ 2. 一个简单的例子——百元买百鸡问题

- 问题描述
  - 已知公鸡 5 元一只，母鸡 3 元一只，小鸡 1 元三只，用 100 元钱买 100 只鸡，问公鸡、母鸡、小鸡各多少只？
- 求解思路
  - 设公鸡、母鸡和小鸡的个数为 x、y、z，则有如下方程组成立：

$$\begin{cases} x + y + z = 100 \\ 5 \times x + 3 \times y + z / 3 = 100 \end{cases}$$

$$\begin{cases} 0 \le x \le 20 \\ 0 \le y \le 33 \\ 0 \le z \le 100 \end{cases}$$

  - 方程组可能有多个解，则需要输出所有满足条件的解。

8.1 概述

■ 2. 一个简单的例子——百元买百鸡问题

□算法实现

设变量 x 表示公鸡的个数，y 表示母鸡的个数，z 表示小鸡的个数，count表示解的个数，算法如下：

算法：百元买百鸡问题
输入：100元钱，100只鸡
输出：所有可行解

1. 初始化解的个数 count = 0；
2. 循环变量 x 从 0~20 循环执行下述操作：
   2.1 循环变量 y 从 0~33 循环执行下述操作：
      2.1.1 z = 100 - x - y；
      2.1.2 如果 5*x + 3*y + z/3 等于 100，则 count++；输出 x、y和z的值；
      2.1.3 y++；
   2.2 x++；
3. 如果 count 等于 0，则输出无解信息；

8.1 概述

- 2. 一个简单的例子——百元买百鸡问题
  - 程序实现
    - 注意到小鸡 1 元三只，在判断总价是否满足方程时要先判断 z 是否是3的倍数。程序如下：

```
void Chicken( )
{
    int x, y, z, count = 0;
    for (x = 0; x <= 20; x++)
        for (y = 0; y <= 33; y++)
        {
            z = 100 - x - y;                          //满足共100只
            if ((z % 3 == 0) && (5 * x + 3 * y + z/3 == 100))  //满足总价
            {
                count++;                              //解的个数加1
                cout<<"公鸡："<<x<<"母鸡："<<y<<"小鸡："<<z<<endl;
            }
        }
    if (count == 0)
        cout<<"问题无解"<<endl;
}
```

21次

34次

0	25	75
4	18	78
8	11	81
12	4	84

时间复杂度？ → $O(n^2)$

中国科学院大学
University of Chinese Academy of Sciences

8.1 概述

■ 3. 最大连续子序列和

- 给定一个含n（n≥1）个整数的序列，要求求出其中最大连续子序列的和。
  - 序列(-2, $\underbrace{11, -4, 13}$, -5, -2)的最大子序列和为20。
  - 序列(-6, 2, 4, -7, $\underbrace{5, 3, 2, -1, 6, -9, 10}$, -2)的最大子序列和为16。
  - 规定一个序列最大连续子序列和至少是0，如果小于0，其结果为0。

8.1 概述

■ 3. 最大连续子序列和

- 解法1：设含有n个整数的序列a[0..n-1]，枚举所有连续子序列a[i..j]。其中任何连续子序列a[i..j]（i≤j，0≤i≤n-1，i≤j≤n-1）求出它的所有元素之和thisSum。
- 通过比较将最大值存放在maxSum中，最后返回maxSum。

$a[0]\ a[1]\ \ldots\ \underbrace{a[i]\ a[i+1]\ \ldots\ a[j]}_{thisSum}\ a[j+1]\ \ldots\ a[n-1]$

$$thisSum \xrightarrow{MAX} maxSum$$

$$\left.\begin{array}{l} i\text{：}0\sim n-1 \\ j\text{：}i\sim n-1 \end{array}\right\} k\text{：}i\sim j$$

8.1 概述

■ 3. 最大连续子序列和

- □解法1：设含有n个整数的序列a[0..n-1]，枚举所有连续子序列a[i..j]。

a[0..5] = {-2, 11, -4, 13, -5, -2}

j↓

	0	1	2	3	4	5	
	-2	11	-4	13	-5	-2	初始序列
i → 0	-2	9	5	18	13	11	
1		11	7	**20**	15	13	
2			-4	9	4	2	
3				13	8	6	
4					-5	-7	
5						-2	

8.1 概述

■ 3. 最大连续子序列和

□解法1：设含有n个整数的序列a[0..n-1]，枚举所有连续子序列a[i..j]。

```
int maxSubSum1(vector<int>&a)                    //解法1
{   int n=a.size();
    int maxsum=0,cursum;
    for (int i=0; i<n; i++)                      //两重循环穷举所有连续子序列
    {   for (int j=i; j<n;j++)
        {   cursum=0;
            for (int k=i; k<=j; k++)             //求a[i..j]子序列元素和cursum
                cursum+=a[k];
            maxsum=max(maxsum, cursum);          //比较求最大连续子序列之和
        }   }
    return maxsum;
}
```

中国科学院大学
University of Chinese Academy of Sciences

8.1 概述

- **3. 最大连续子序列和**
  - 解法1：设含有n个整数的序列a[0..n-1]，枚举所有连续子序列a[i..j]。
    - maxSubSum1(a, n)算法中用了三重循环，所以有：

$$\mathbf{T(n)}=\sum_{i=0}^{n-1}\sum_{j=i}^{n-1}\sum_{k=i}^{j}1=\sum_{i=0}^{n-1}\sum_{j=i}^{n-1}(j-i+1)=\frac{1}{2}\sum_{i=0}^{n-1}(n-i)(n-i+1)=\mathbf{O(n^3)}$$

8.1 概述

- **3. 最大连续子序列和**
  - 解法2：优化点 ⇨ 避免起始下标i开始的子序列的重复计算。
    - 在求两个相邻子序列和时，它们之间是关联的。
    - 例如a[0..3]子序列和=a[0]+a[1]+a[2]+a[3]，a[0..4]子序列和=a[0]+a[1]+a[2]+a[3]+a[4]，在前者计算出来后，求后者时只需在前者基础上加以a[4]即可，没有必须每次都重复计算。从而提高了算法效率。

- 用Sum(a[i..j])表示子序列a[i..j]元素和，初始时置Sum(a[i..j])=0，显然有如下递推关系：

  Sum(a[i..j])= Sum(a[i..j-1])+a[j]　当j≥i时

- 连续求a[i..j]子序列和（j=i, i+1, …, n-1）时没有必要使用循环变量为k的第3重循环。

8.1 概述

■ 3. 最大连续子序列和

- □解法2：优化点 ⇨ 避免起始下标i开始的子序列的重复计算。

Sum[1]=Sum[0]+a[1]=9

j↓	0	1	2	3	4	5
初始序列	-2	11	-4	13	-5	-2
i → 0	-2	9	5	18	13	11
1		11	7	20	15	13
2			-4	9	4	2
3				13	8	6
4					-5	-7
5						-2

8.1 概述

■ 3. 最大连续子序列和

□解法2：优化点 ⇨ 避免起始下标i开始的子序列的重复计算。

```
int maxSubSum2(vector<int>& a)                    //解法2
{   int n=a.size();
    int maxsum=0, cursum;
    for (int i=0; i<n; i++)
    {   cursum=0;
        for (int j=i; j<n; j++)                 //连续求a[i..j]子序列元素和cursum
        {   cursum+=a[j];
            maxsum=max(maxsum, cursum);          //比较求最大maxsum
        }
    }
    return maxsum;
}
```

maxSubSum2(a, n)算法中只有两重循环，容易求出$T(n)=O(n^2)$。

8.1 概述

■ 3. 最大连续子序列和

- 解法3：优化点 ⇨ maxsum至少为0。
  - 如果扫描中遇到负数，当前子序列和thisSum将会减小，若thisSum为负数，表明前面已经扫描的那个子序列可以抛弃了，则放弃这个子序列，重新开始下一个子序列的分析，并置thisSum为0。
  - 若这个子序列和thisSum不断增加，那么最大子序列和maxSum也不断增加。

8.1 概述

■ 3. 最大连续子序列和

□解法3：优化点 ⇨ maxsum至少为0。

maxsum=0, cursum=0（cursum表示以a[i]结尾的最大和）

a[0..5]={-2， 11， -4， 13， -5， -2}

-2：
cursum=0+(-2)=-2
maxsum=0
cursum<0从头开始
cursum=0

11：
cursum=0+11=11
maxsum=11

-4：
cursum=11+(-4)=7
maxsum=11

13：
cursum=7+13=20
maxsum=20

-5：
cursum=20+(-5)=15
maxsum=20

-2：
cursum=15+(-2)=13
maxsum=20

maxsum=20

8.1 概述

■ 3. 最大连续子序列和

□解法3：优化点 ⇨ maxsum至少为0。

```
int maxSubSum3(vector<int>& a)              //解法3
{   int n=a.size();
    int maxsum=0, cursum=0;
    for (int i=0; i<n; i++)
    {   cursum+=a[i];                       //cursum表示以a[i]结尾的最大和
        maxsum=max(maxsum, cursum);         //比较求最大maxsum
        if(cursum<0)                        //若cursum<0，从下一个位置开始
            cursum=0;
    }
    return  maxsum;
}
```

T(n)=O(n)

第八章 蛮力法

- 8.1 概述
- 8.2 串匹配问题

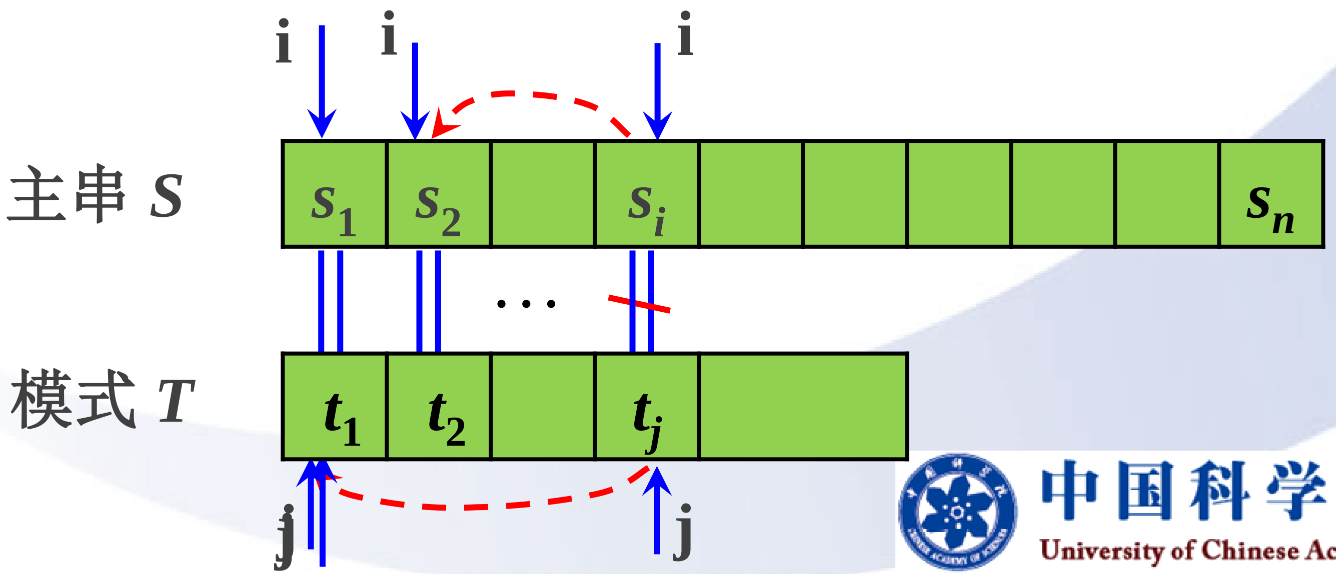
8.2 串匹配问题

- **1. 问题描述**
  - 给定两个字符串*S*和*T*，在主串*S*中查找子串*T*的过程，称为串匹配，又称为模式匹配，*T*称为模式。
  - 如果匹配成功，返回 *T* 在 *S* 中的位置；否则返回 0。
- **2. 特点**
  ① 问题规模通常很大，常常在大量信息中进行匹配，因此算法的一次执行时间不容忽视
  ② 串匹配操作经常被调用，执行频率高，因此算法改进所取得的积累效益大

中国科学院大学
University of Chinese Academy of Sciences

8.2 串匹配问题

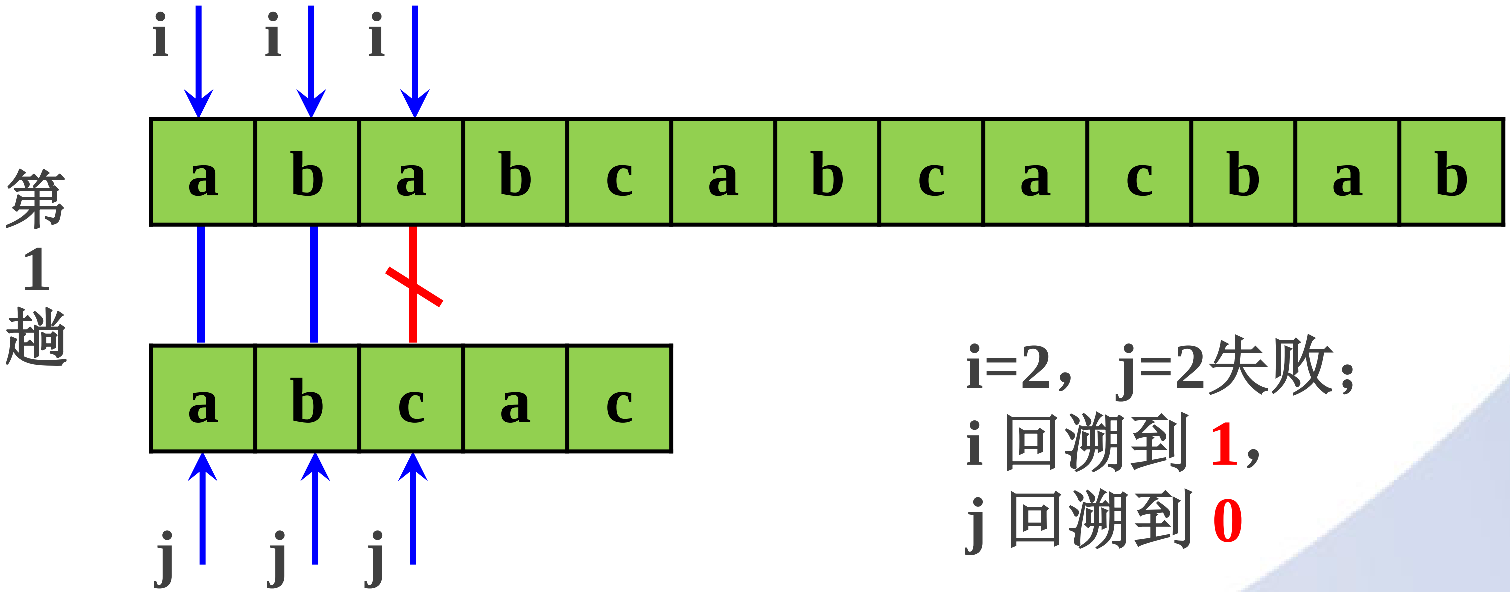
- 3. 朴素的模式匹配算法（BF算法）
  - 从主串 *S* 的第一个字符开始和模式 *T* 的第一个字符进行比较，若相等，则继续比较两者的后续字符；
  - 若不相等，则从主串 *S* 的第二个字符开始和模式 *T*的第一个字符进行比较，
  - 重复上述过程，若 *T* 中的字符全部比较完毕，则说明本趟匹配成功；若 *S* 中的字符全部比较完毕，则匹配失败。



8.2 串匹配问题

■ 3. 朴素的模式匹配算法（BF算法）

□例：主串 *S* = "ababcabcacbab"，模式 *T* ="abcac"



i=2，j=2失败；

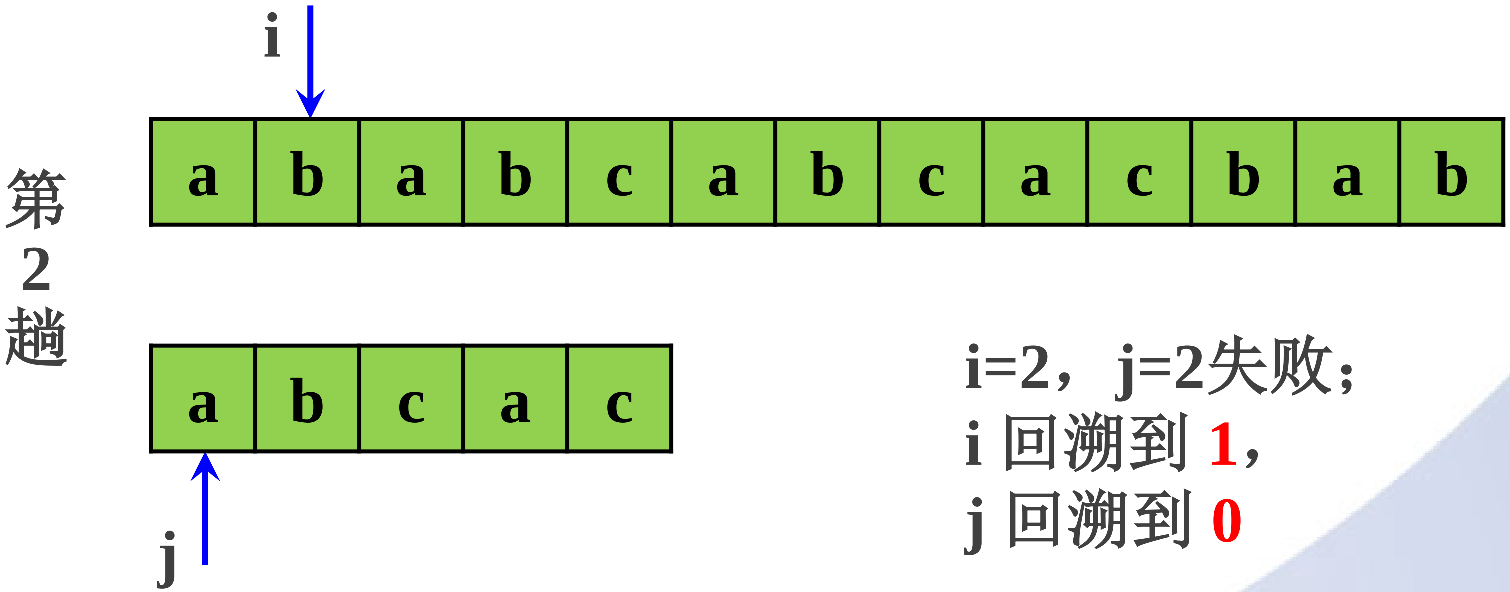
i 回溯到 1，

j 回溯到 0

8.2 串匹配问题

■ 3. 朴素的模式匹配算法（BF算法）

□ 例：主串 *S* = "ababcabcacbab"，模式 *T* ="abcac"

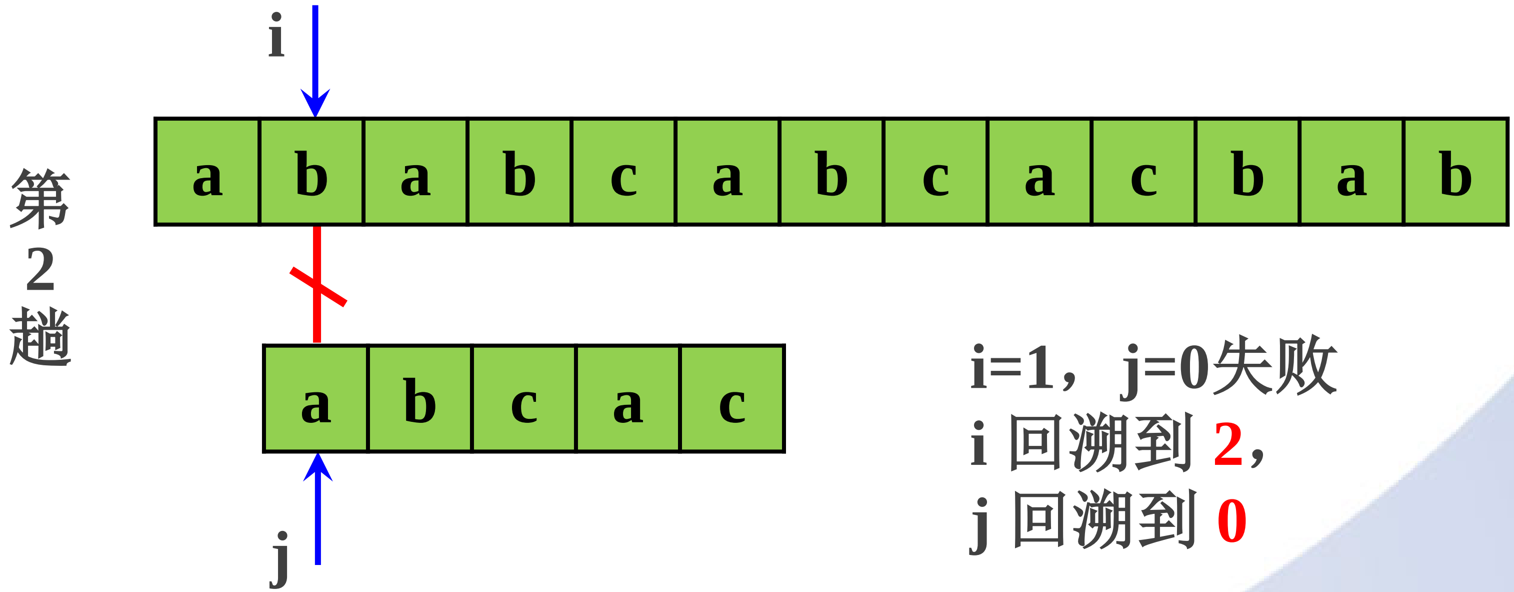


i=2，j=2失败；
i 回溯到 1，
j 回溯到 0

8.2 串匹配问题

■ 3. 朴素的模式匹配算法（BF算法）

□例：主串 S = "ababcabcacbab"，模式 T ="abcac"



i=1，j=0失败

i 回溯到 2，

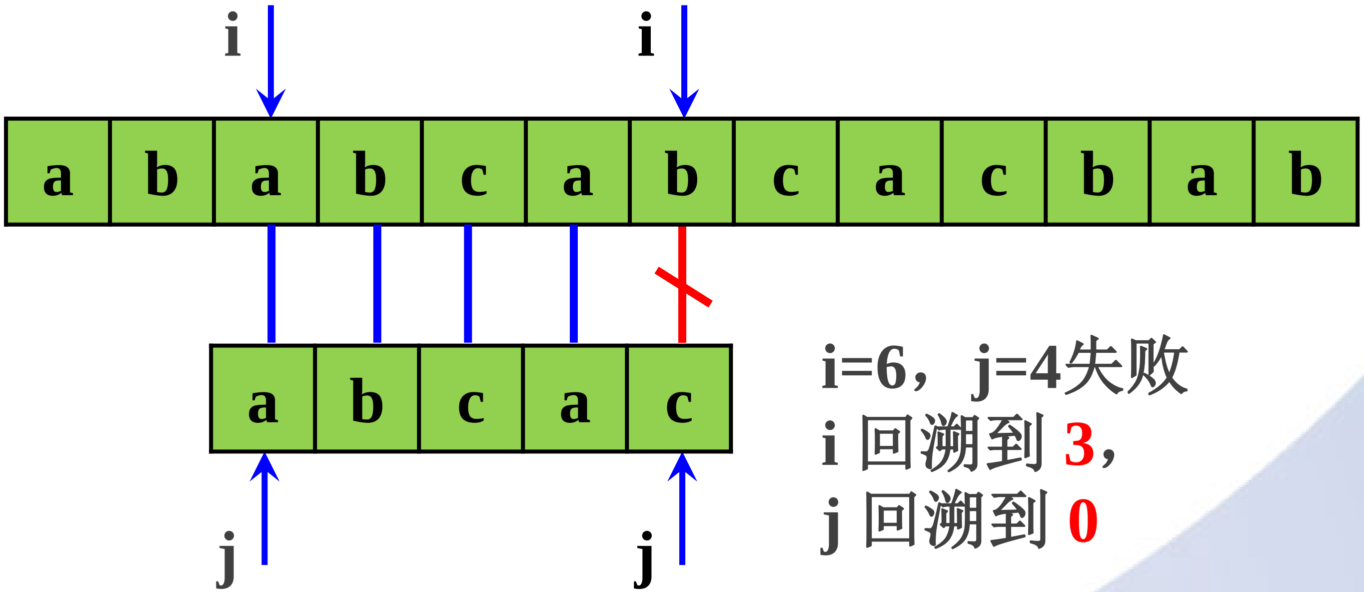
j 回溯到 0

8.2 串匹配问题

■ 3. 朴素的模式匹配算法（BF算法）

□例：主串 *S* = "ababcabcacbab"，模式 *T* ="abcac"

第3趟



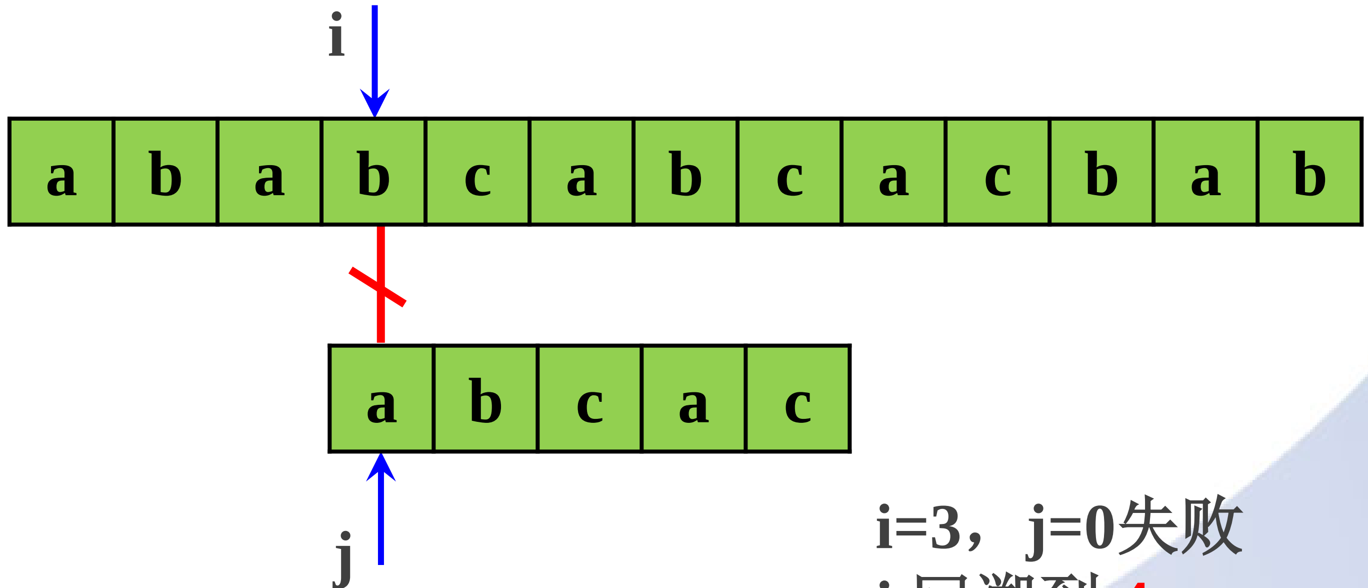
i=6，j=4失败
i 回溯到 **3**，
j 回溯到 **0**

8.2 串匹配问题

■ 3. 朴素的模式匹配算法（BF算法）

□ 例：主串 *S* = "ababcabcacbab"，模式 *T* ="abcac"

第4趟



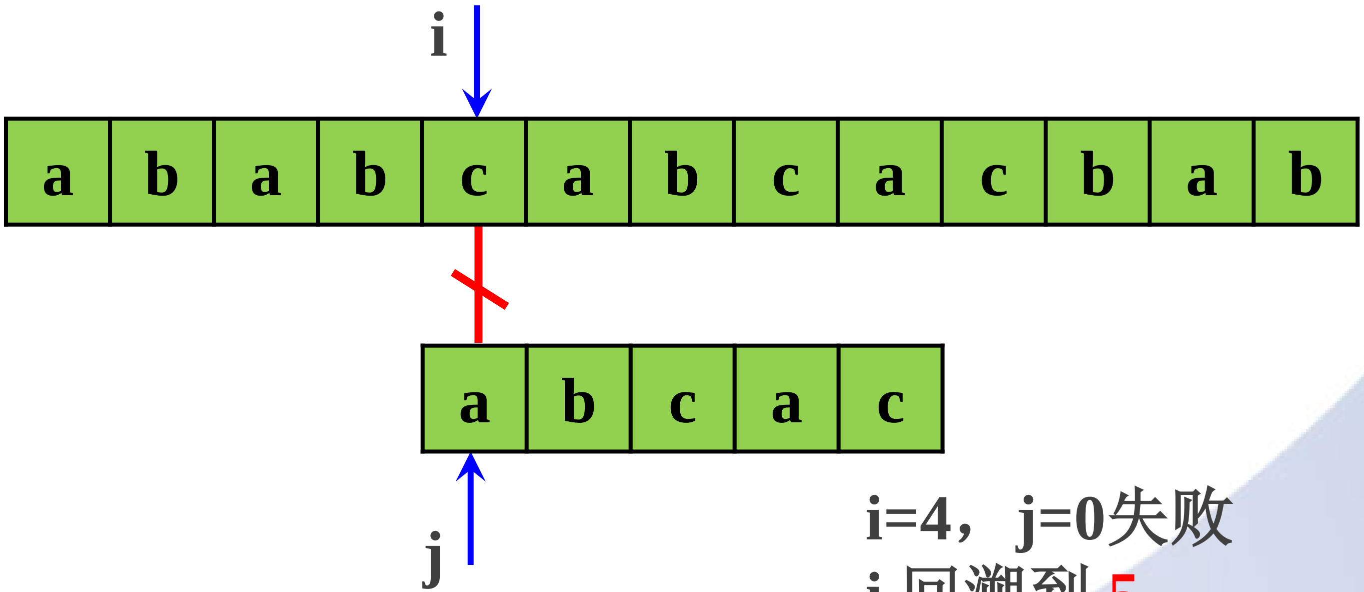
i=3，j=0失败
i 回溯到 **4**,
j 回溯到 **0**

8.2 串匹配问题

■ 3. 朴素的模式匹配算法（BF算法）

□例：主串 *S* = "ababcabcacbab"，模式 *T* ="abcac"

第5趟



i=4，j=0失败

i 回溯到 5，

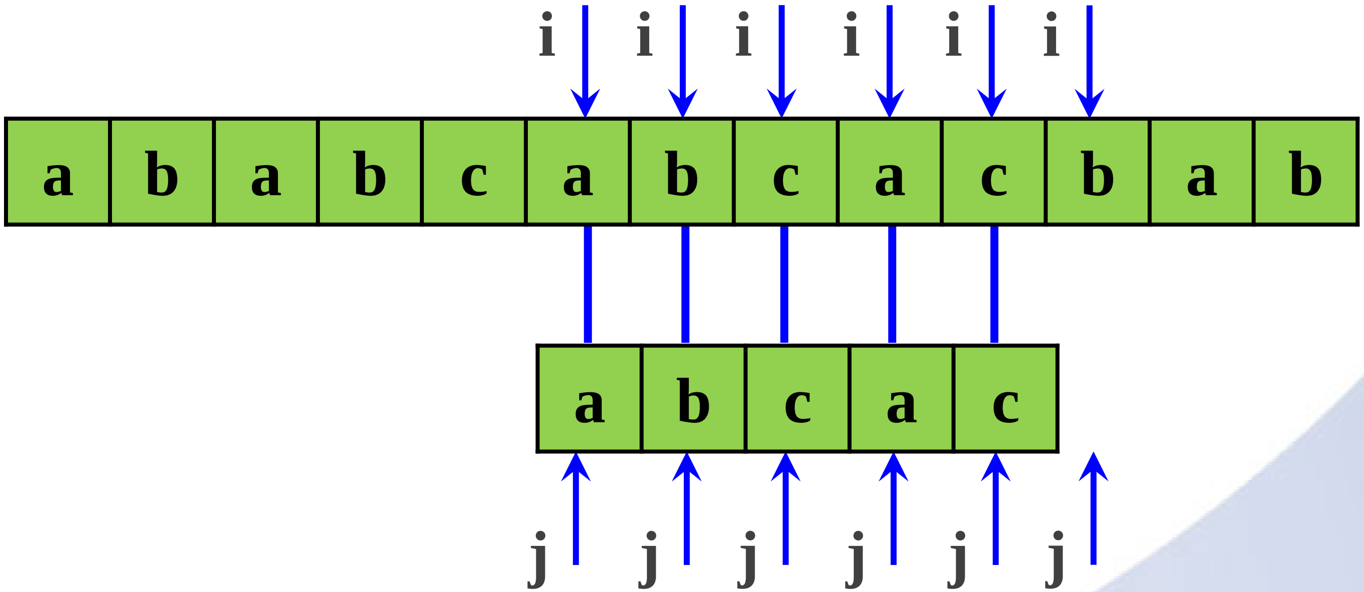
j 回溯到 0

8.2 串匹配问题

■ 3. 朴素的模式匹配算法（BF算法）

□例：主串 *S* = "ababcabcacbab"，模式 *T* ="abcac"

第6趟



i=10，j=5，*T*中全部字符都比较完毕，匹配成功

8.2 串匹配问题

- 3. 朴素的模式匹配算法（BF算法）
  - 算法1，设字符数组 *S* 存放主串，字符数组 *T* 存放模式，BF算法如下：

算法：串匹配算法BF

输入：主串 *S*，模式 *T*

输出：*T* 在 *S* 中的位置

1. 初始化主串比较的开始位置 index = 0;
2. 在串 *S* 和串 *T* 中设置比较的起始下标 i = 0, j = 0;
3. 重复下述操作，直到 *S* 或 *T* 的所有字符均比较完毕：
   3.1 如果 S[i] 等于 T[j]，则继续比较 *S* 和 *T* 的下一对字符；
   3.2 否则，下一趟匹配的开始位置 index++，回溯下标 i = index，j = 0;
4. 如果 *T* 中所有字符均比较完，则返回匹配的开始位置；否则返回0;

8.2 串匹配问题

■ **3. 朴素的模式匹配算法（BF算法）**

□ 算法分析

- 设主串S长度为n, 模式T长度为m，
- 在匹配成功的情况下，考虑最坏情况，即每趟不成功的匹配都发生在串T的最后一个字符。
- 设匹配成功发生在 s_i 处，则$(i\text{-}1)$次不成功的匹配共比较$(i\text{-}1)\times m$次，第i次成功比较了m次，所以总共比较了$i\times m$次，
- 则平均比较次数为：

$$\sum_{i=1}^{n-m+1} p_i \times (i \times m) = \sum_{i=1}^{n-m+1} \frac{1}{n-m+1} \times (i \times m) = \frac{m(n-m+2)}{2} = O(m \times n)$$

8.2 串匹配问题

■ 3. 朴素的模式匹配算法（BF算法）

□算法实现

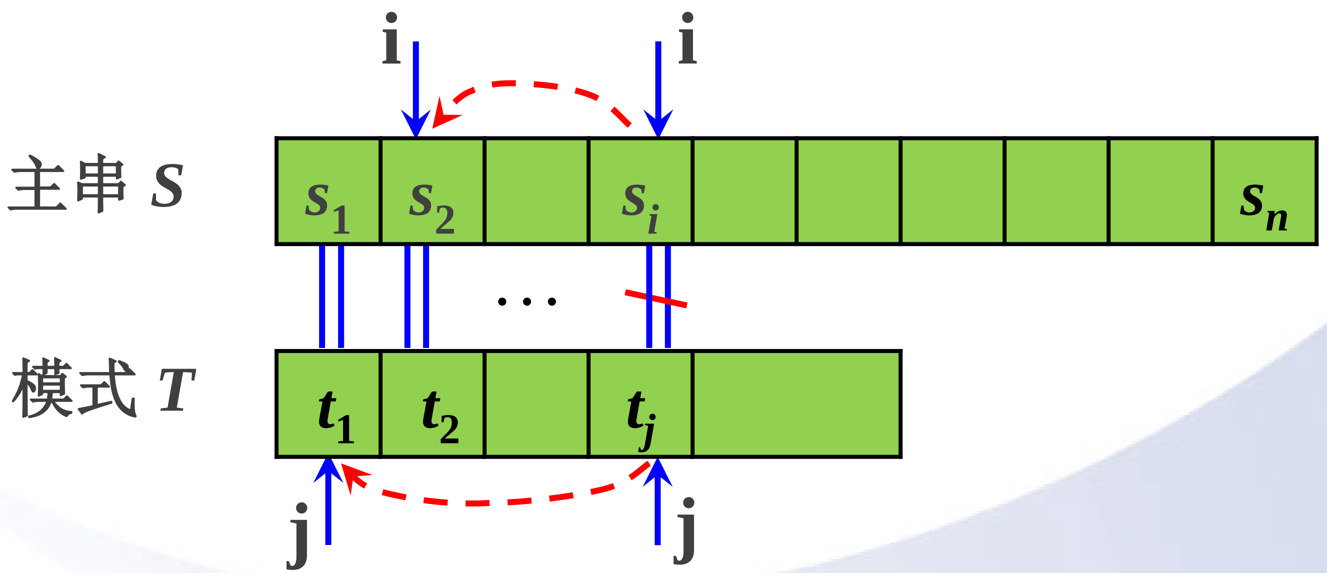
```
int BF(char S[ ], char T[ ])
{
    int index = 0, i = 0, j = 0;
    while ((S[i] != '\0') && (T[j] != '\0'))
    {
        if (S[i] == T[j]) { i++; j++; }
        else { index++; i = index; j = 0; }          // i 和 j 分别回溯
    }
    if (T[j] == '\0') return index + 1;
    else return 0;
}
```

8.2 串匹配问题

■ 3. 朴素的模式匹配算法（BF算法）

□ 问题分析

➢ 在每趟匹配不成功时存在大量回溯，没有利用已经部分匹配的结果

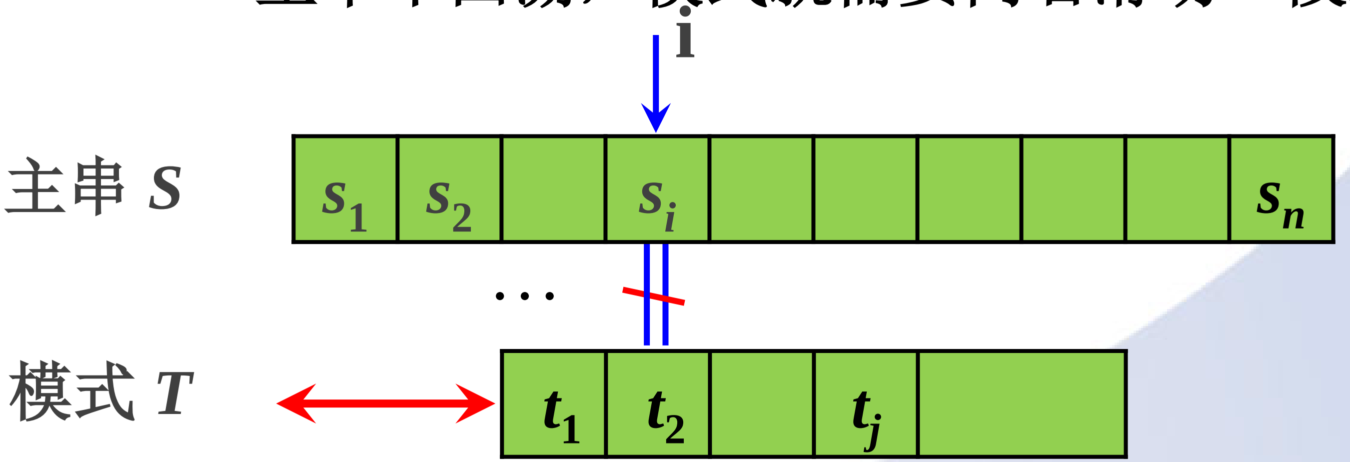


8.2 串匹配问题

■ 4. KMP算法

□基本思想

- ➢主串不进行回溯，模式 *T* 要回溯到某一个字符
- ➢如何在匹配不成功时主串不回溯？
  - ✓主串不回溯，模式就需要向右滑动一段距离



- ➢如何确定模式的滑动距离？

8.2 串匹配问题

第1趟

i

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

a	b	c	a	c

j

i=2， j=2失败；

∵ s[1]=t[1]; t[0]≠t[1]

∴ t[0]≠s[1]

第2趟

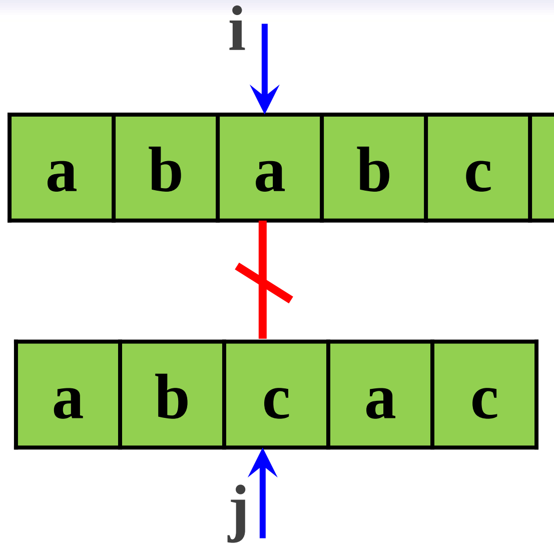
a	b	a	b	c	a	b	c	a	c	b	a	b

a	b	c	a	c

t[0]≠s[1]

8.2 串匹配问题

第1趟

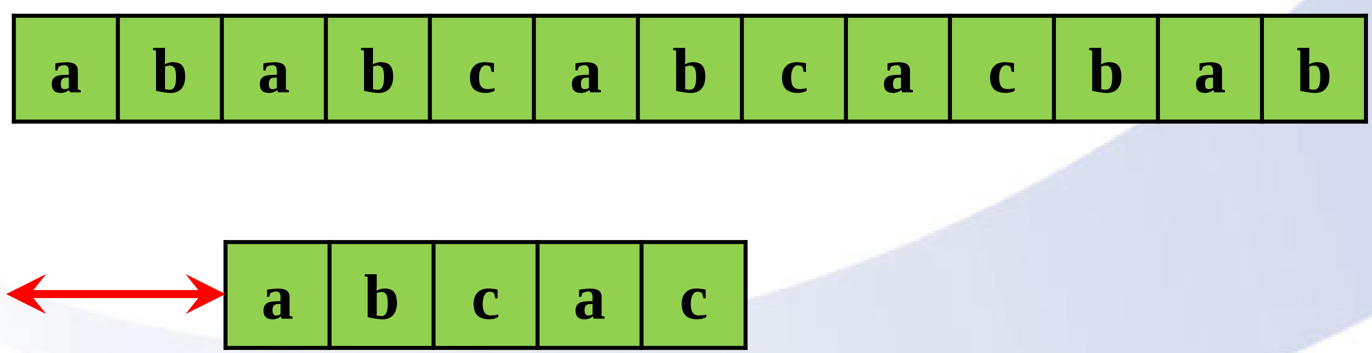


i=2，j=2失败；

∵s[1]=t[1]; t[0]≠t[1]

∴t[0]≠s[1]

第3趟



8.2 串匹配问题

第3趟

i

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

a	b	c	a	c

j

i=6，j=4失败

∵ s[3]=t[1]; t[0]≠t[1]

∴ t[0]≠s[3]

第4趟

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

a	b	c	a	c

t[0]≠s[3]

8.2 串匹配问题

第3趟

i

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

a	b	c	a	c

j

i=6， j=4失败

∵ s[4]=t[2]; t[0]≠t[2]

∴ t[0]≠s[4]

第5趟

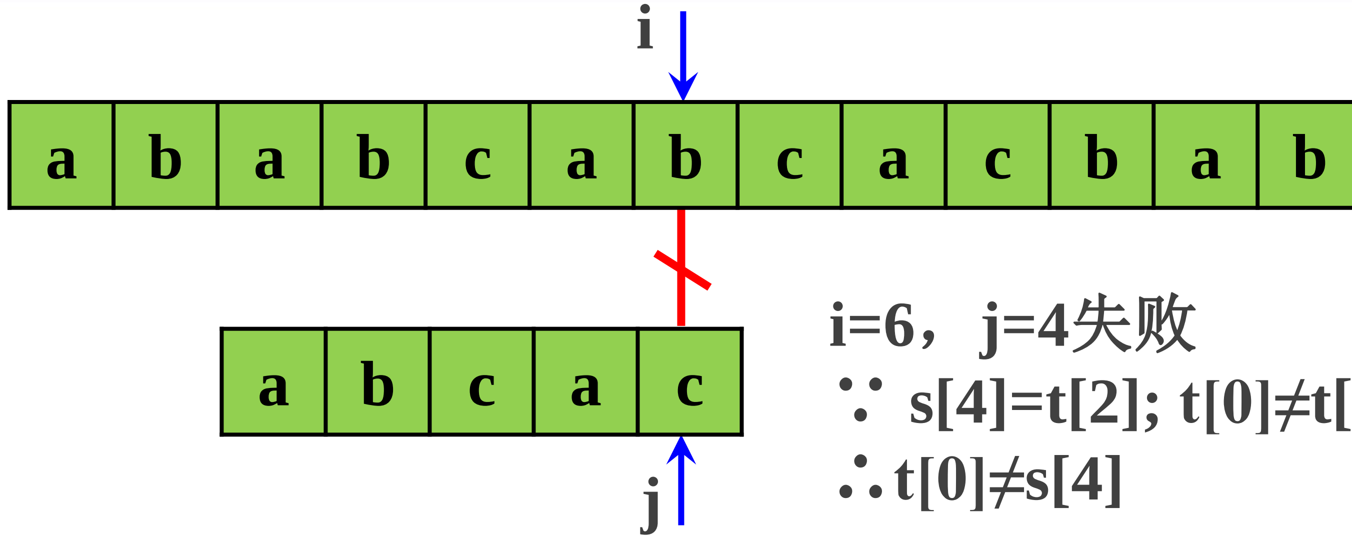
a	b	a	b	c	a	b	c	a	c	b	a	b

a	b	c	a	c

t[0]≠s[4]

8.2 串匹配问题

第3趟

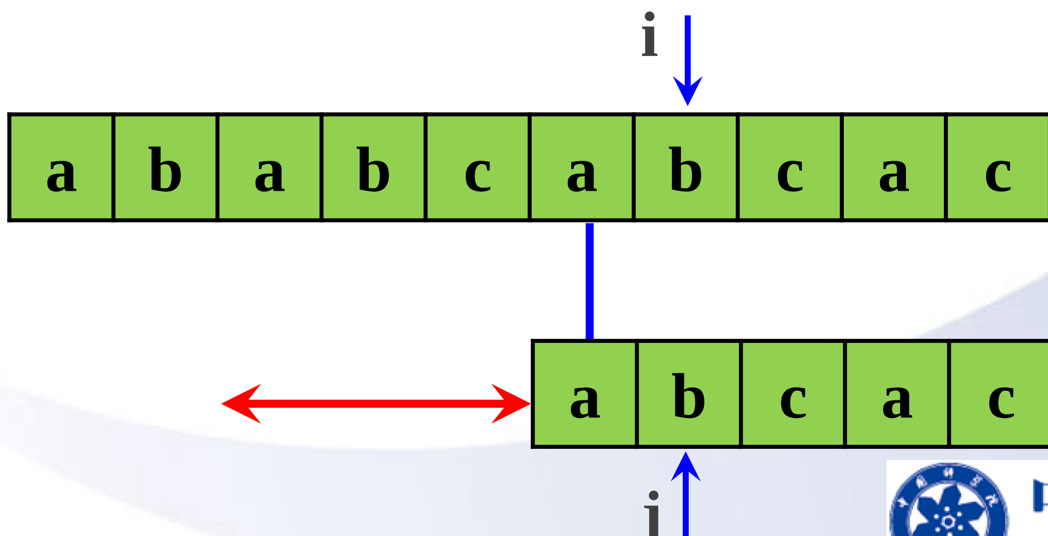


i=6，j=4失败

∵ s[4]=t[2]; t[0]≠t[2]

∴ t[0]≠s[4]

第6趟



i=6，j=4失败

∵ s[5]=t[3]; t[0]=t[3]

∴ t[0]=s[5]

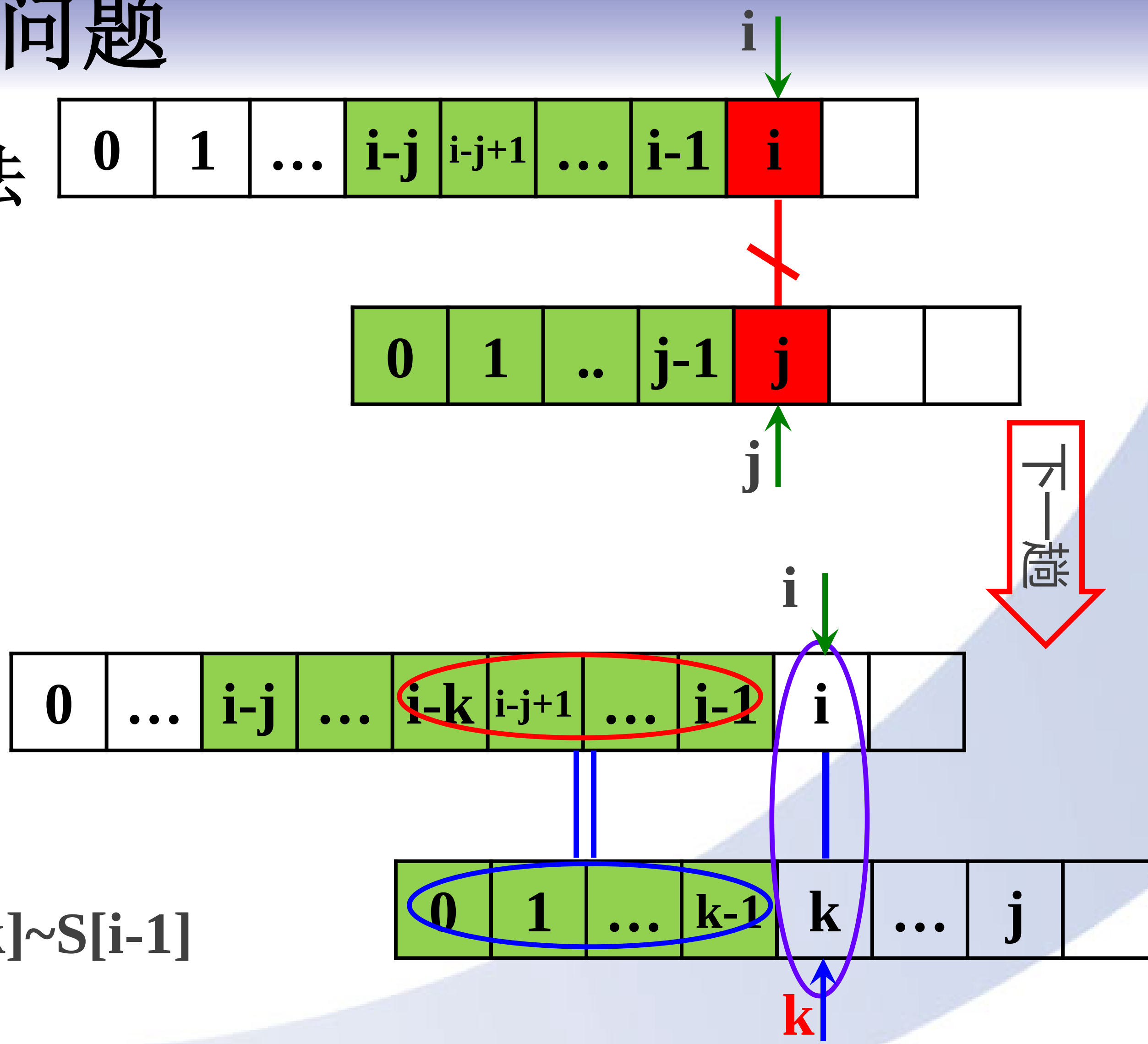
8.2 串匹配问题

- **4. KMP算法**
  - 基本思想
    - 如何确定模式的滑动距离？
    - 结论： i 可以不回溯，模式向右滑动到新的比较起点 k
    - 如何由当前部分匹配结果确定模式向右滑动的新比较起点 k？

8.2 串匹配问题

■ 4. KMP算法

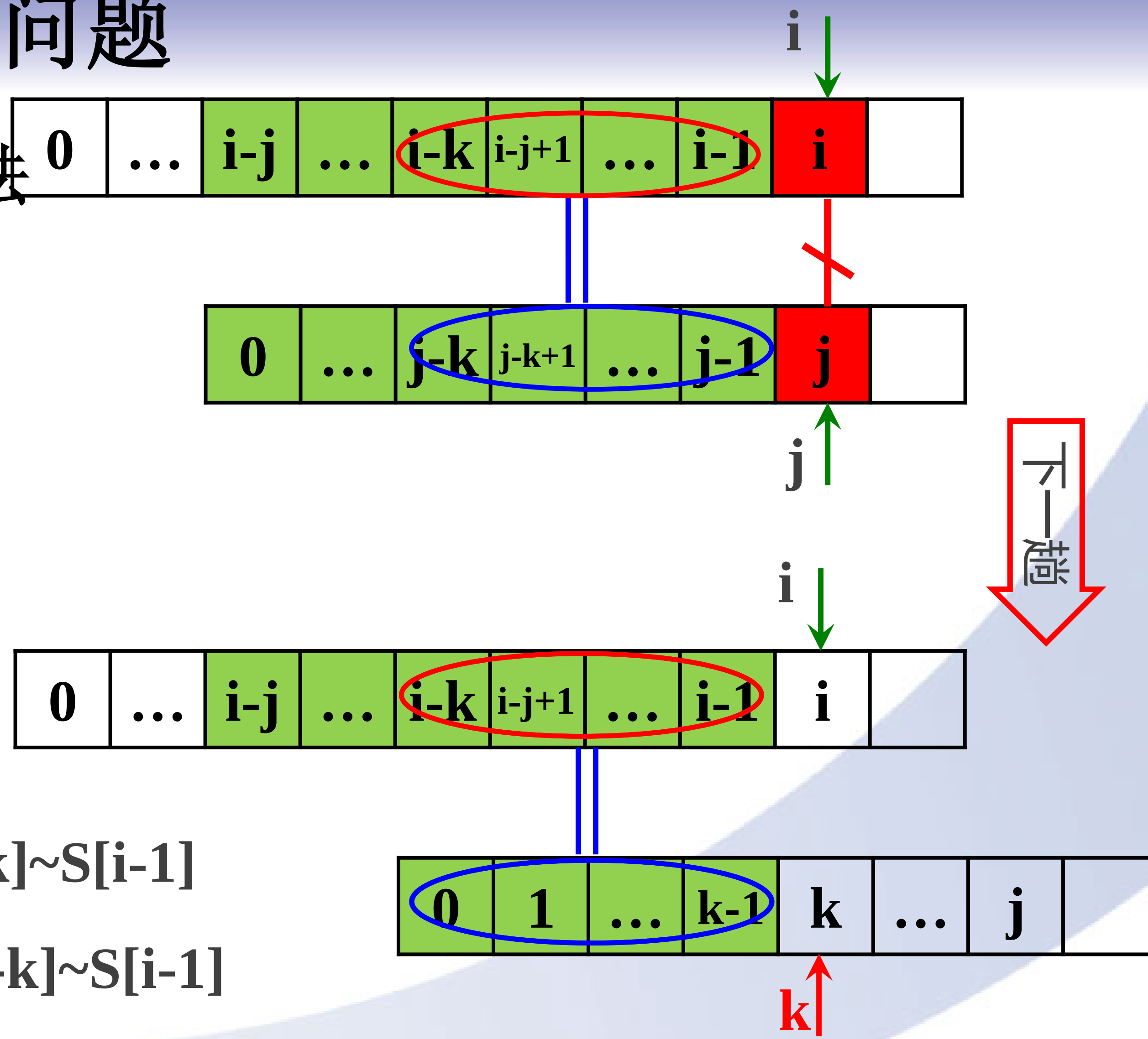
□基本思想



(1) T[0]~T[k-1] = S[i-k]~S[i-1]

8.2 串匹配问题

- **4. KMP算法**
  - 基本思想



(1) T[0]~T[k-1] = S[i-k]~S[i-1]

(2) T[j-k]~T[j-1] = S[i-k]~S[i-1]

=>T[0]~T[k-1] = T[j-k]~T[j-1]

8.2 串匹配问题

■ 4. KMP算法

□基本思想

➢T[0] ~ T[k-1] = T[j-k] ~ T[j-1] 说明了什么？

① *k* 与 *j* 具有函数关系，由当前失配位置 *j*，可以计算出滑动位置 *k*

② 滑动位置 *k* 仅与模式串 *T* 有关

➢T[0] ~ T[k-1] = T[j-k] ~ T[j-1] 的物理意义是什么？

长度为*k*的前缀

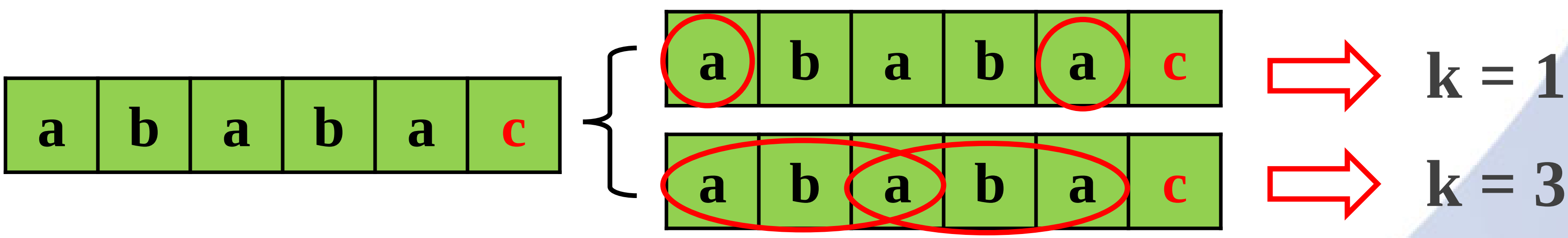
长度为*k*的后缀

8.2 串匹配问题

■ 4. KMP算法

□基本思想

➢T[0] ~ T[j]中前缀和后缀相等的真子串唯一吗?

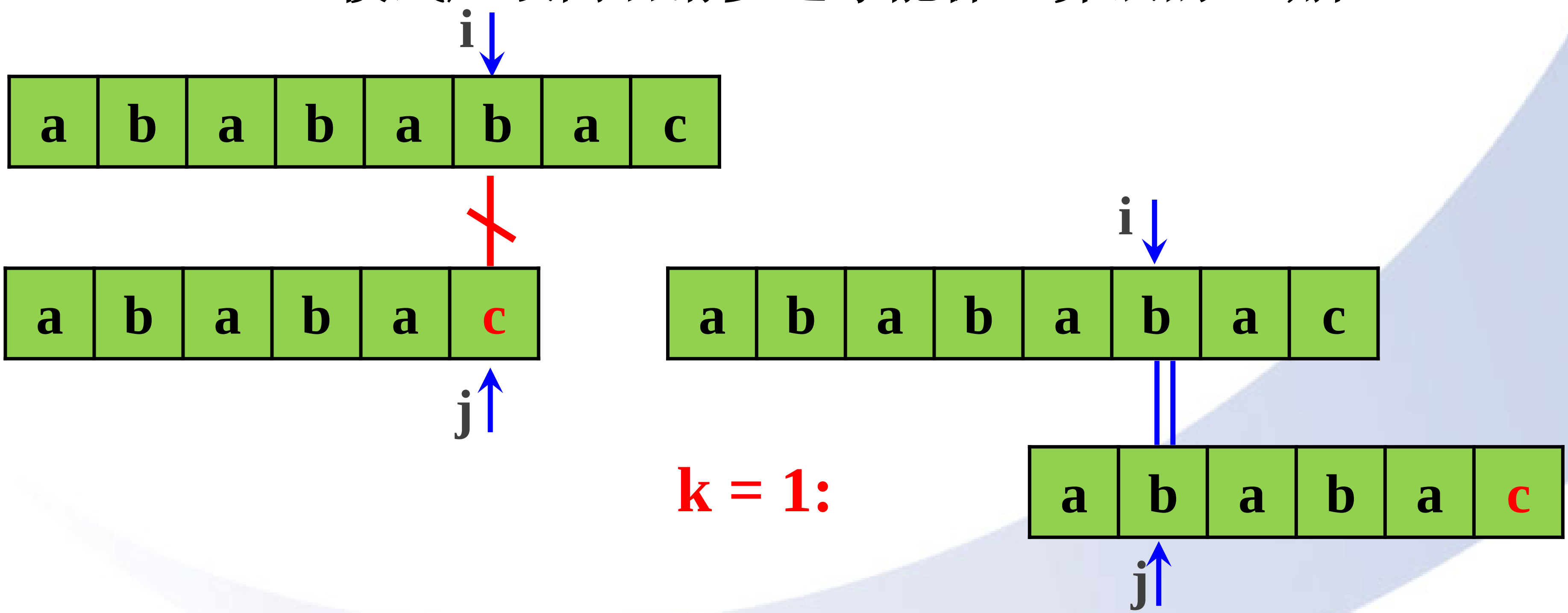


8.2 串匹配问题

■ 4. KMP算法

□基本思想

➢模式应该向右滑多远才能保证算法的正确性?

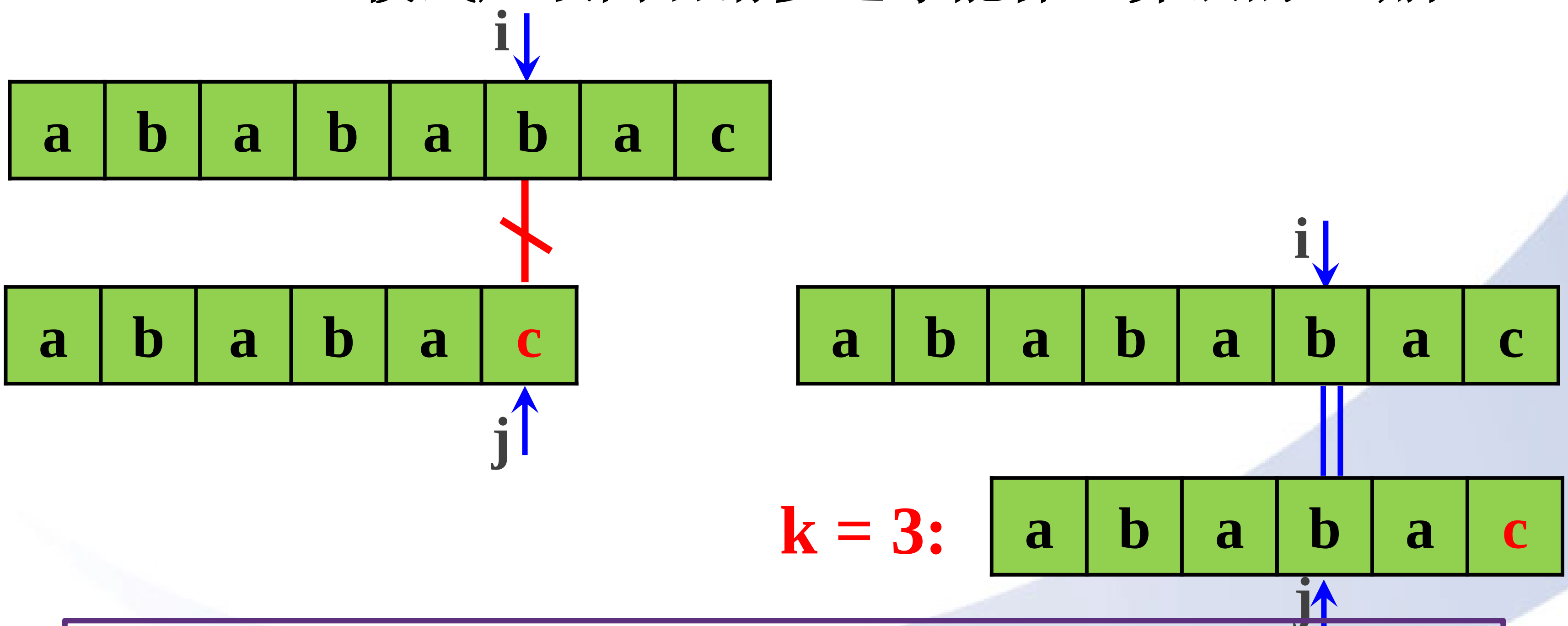


8.2 串匹配问题

■ 4. KMP算法

□基本思想

➢模式应该向右滑多远才能保证算法的正确性?



$$\max\{k \mid 1 \le k < j \text{ 且 } T[0] \ldots T[k-1] = T[j-k] \ldots T[j-1]\}$$

8.2 串匹配问题

- **4. KMP算法**
  - **next值的计算**
    - 设next[j]表示在匹配过程中与T[j]比较不相等时，下标 j 的回溯位置

$$
next[j]=\begin{cases} -1 & j = 0 \\ \max\{k \mid 1 \le k < j \text{ 且 } T[0] \ldots T[k-1] = T[j-k] \ldots T[j-1]\} & \text{集合非空} \\ 0 & \text{其它情况} \end{cases}
$$

8.2 串匹配问题

■ 4. KMP算法

□next值的计算

➢设next[j]表示在匹配过程中与T[j]比较不相等时，下标 j 的回溯位置

下标：	0	1	2	3	4
模 式 串 T：	a	b	a	b	c
k = next[j]：	-1	0	0	1	2

j=0时, k＝-1

j=1时, k＝0

j=2时, T[0]≠T[1]，因此，k = 0

j=3时, T[0]＝T[2]，T[0]T[1] ≠T[1]T[2]，因此，k = 1

j=4时, T[0] ≠ T[3]，T[0]T[1] = T[2]T[3]，T[0]T[1]T[2]≠T[1]T[2]T[3]
因此，k = 2

8.2 串匹配问题

■ 4. KMP算法

□next值的计算

➢模式T="abaababc"的next值计算

j	t[j]前所有字符	t[j]前开头的字符	t[j]字符前的字符	next[j]
0	空			-1
1	a			0
2	ab	a	b	0
3	aba	a, ab	a, ba	1
4	abaa	a, ab,aba	a, aa, baa	1
5	abaab	a, ab, aba, abaa	b, ab, aab, baab	2
6	abaaba	a, ab, aba, abaa, abaab	a, ba, aba, aaba, baaba	3
7	abaabab	a, ab, aba, abaa, abaab, abaaba	b, ab, bab, abab, aabab, baabab	2

8.2 串匹配问题

■ 4. KMP算法

□next值的计算

- ➢设模式的长度为 ***m***，用蛮力法求解 **KMP**算法中的 **next** 值时，最坏情况下的时间代价是：

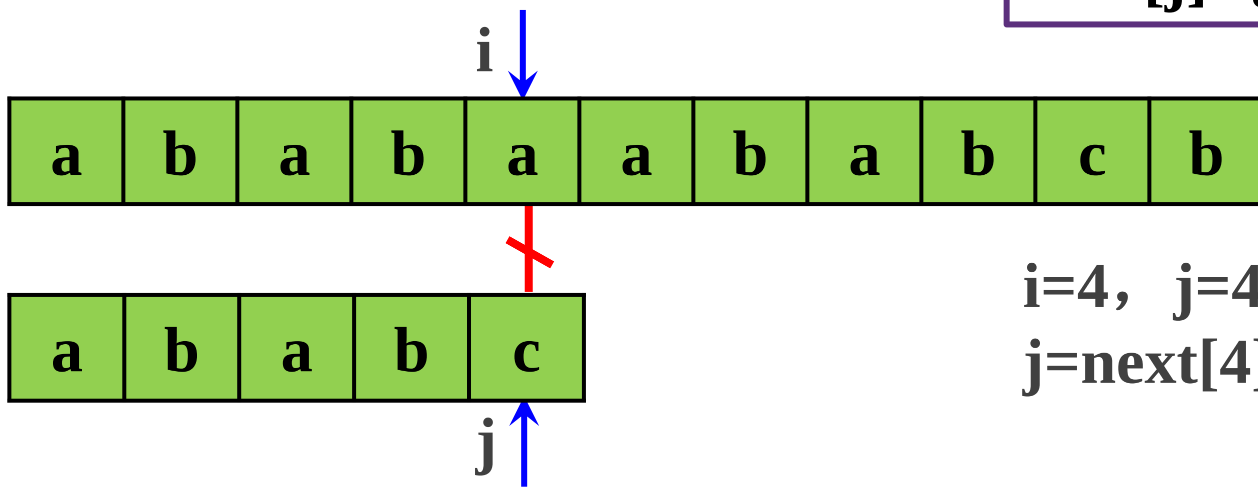
$$\sum_{j=1}^{m-1}(j-1)=\frac{(m-1)(m-2)}{2}=O(m^2)$$

中国科学院大学 University of Chinese Academy of Sciences

8.2 串匹配问题

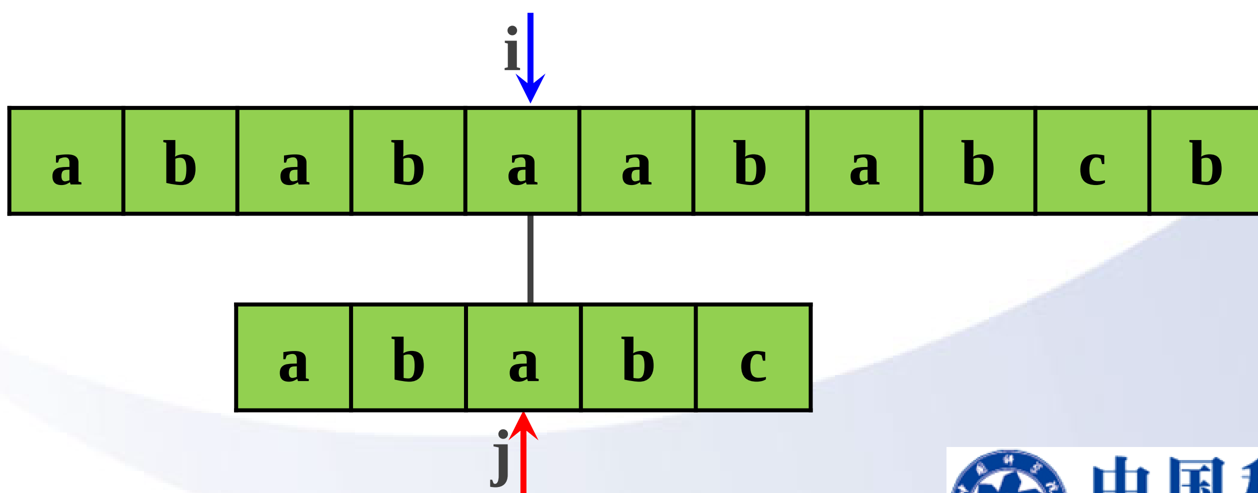
next[j]={-1, 0, 0, 1, 2}

第1趟



i=4， j=4失败；
j=next[4]=2

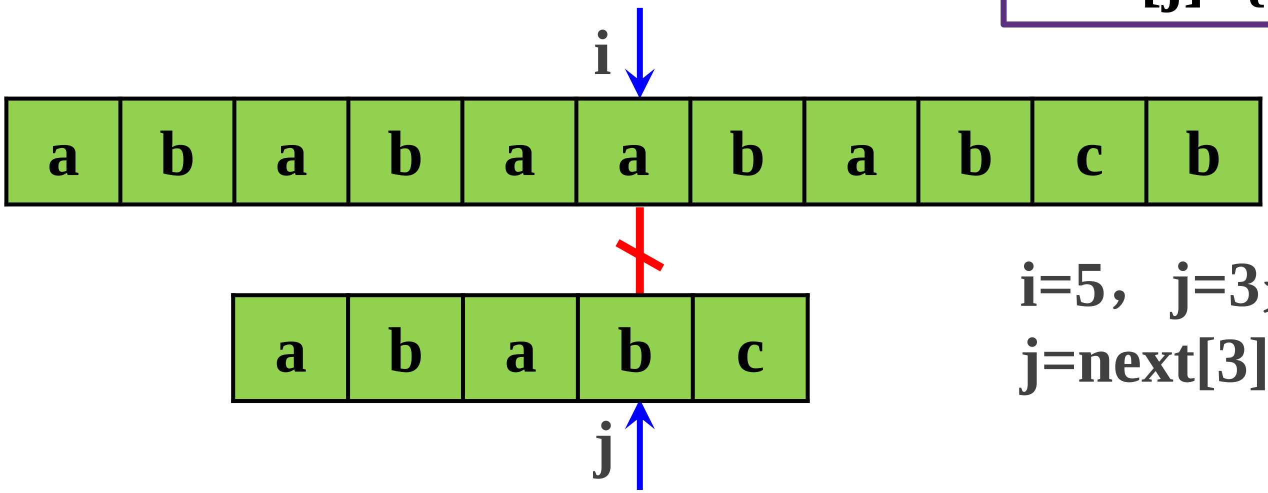
第2趟



8.2 串匹配问题

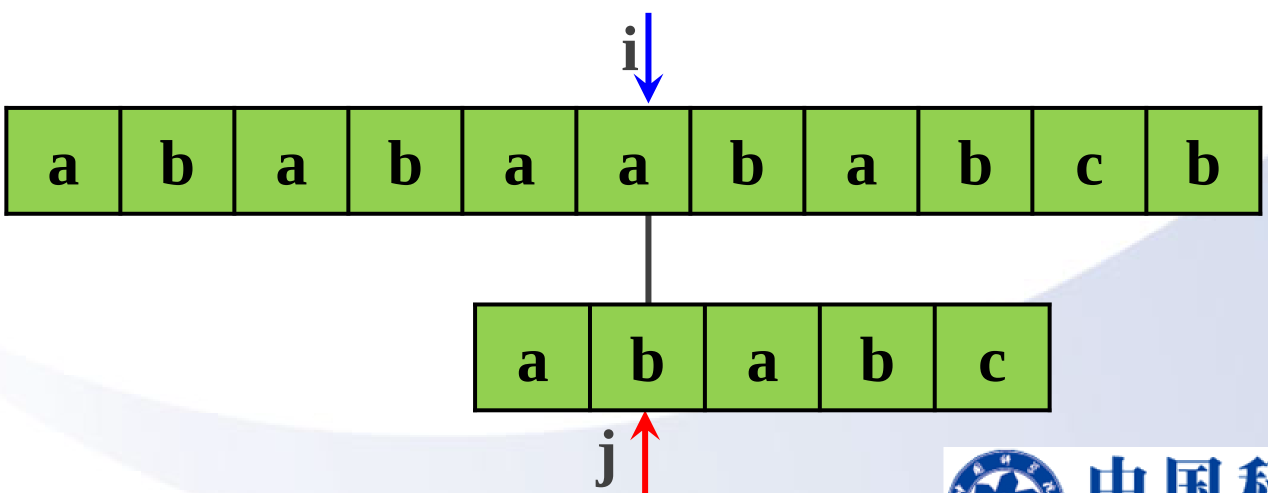
next[j]={-1, 0, 0, 1, 2}

第2趟



i=5，j=3失败；
j=next[3]=1

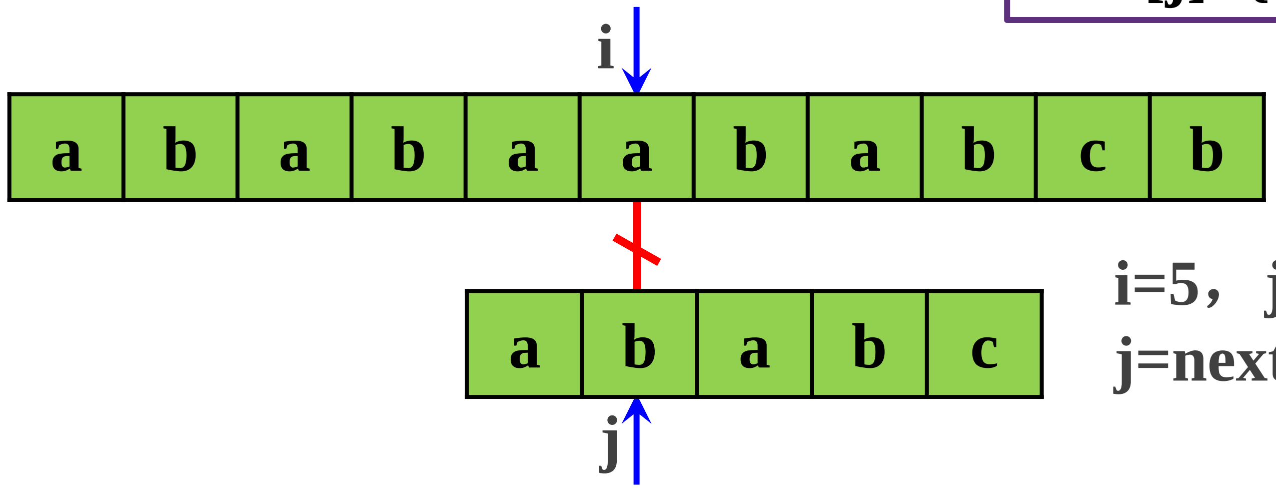
第3趟



8.2 串匹配问题

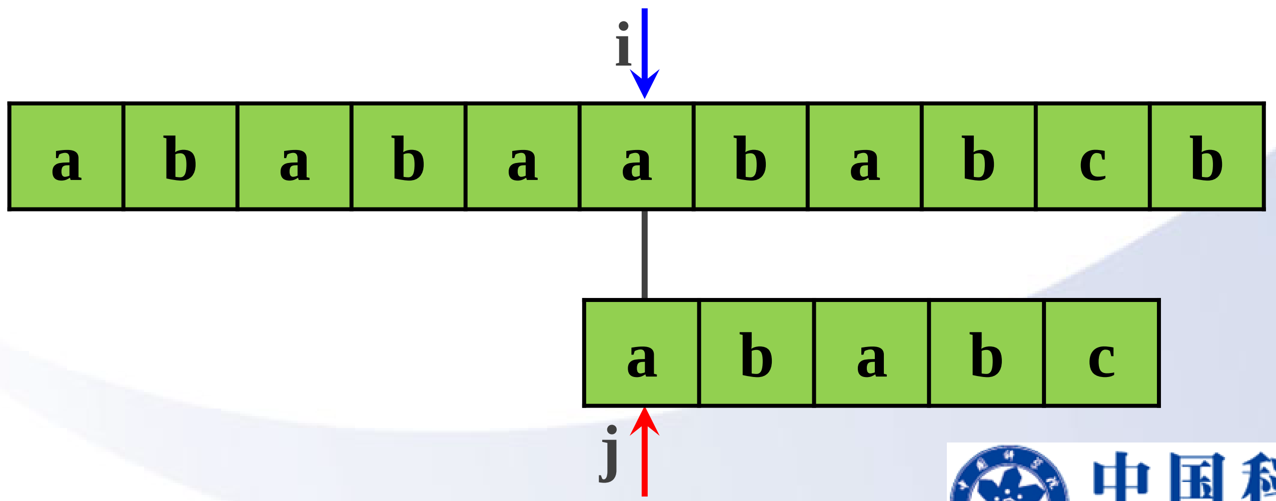
next[j]={-1, 0, 0, 1, 2}

第3趟



i=5， j=1失败；
j=next[1]=0

第4趟



8.2 串匹配问题

■ 4. KMP算法

□算法实现

➢设字符数组 *S* 存放主串，字符数组 *T* 存放模式，在求得了模式 *T* 的next值后，KMP算法如下：

算法：串匹配算法KMP

输入：主串 *S*，模式 *T*

输出：*T* 在 *S* 中的位置

1. 在串 *S* 和串 *T* 中分别设置比较的起始下标 i = 0, j = 0；
2. 重复下述操作，直到 *S* 或 *T* 的所有字符均比较完毕：
   2.1 如果 S[i] 等于 T[j]，则继续比较 S 和 T 的下一对字符；
   2.2 否则，将下标 j 回溯到 next[j] 位置，即 j = next[j]；
   2.3 如果 j 等于 -1，则将下标 i 和 j 分别加 1，准备下一趟比较；
3. 如果 *T* 中所有字符均比较完毕，则返回本趟匹配的开始位置；否则返回0；

8.2 串匹配问题

■ 4. KMP算法

□程序实现

```
void GetNext(char T[ ], int next[ ])
{
    int i, j, len;
    next[0] = -1;
    for (j = 1; T[j]!='\0'; j++)              //依次求next[j]
    {
        for (len = j - 1; len >= 1; len--)      //相等子串的最大长度为j-1
        {
            for (i = 0; i < len; i++)            //比较T[0]~T[len-1]与T[j-len]~T[j-1]
                if (T[i] != T[j-len+i]) break;
            if (i == len) { next[j] = len; break;}
        }
        if (len < 1) next[j] = 0;              //其他情况，无相等子串
    }
}
```

8.2 串匹配问题

■ 4. KMP算法

□程序实现

```
int KMP(char S[ ], char T[ ])            //求T在S中的序号
{
    int i = 0, j = 0,  next[80];             //假定模式最长为80个字符
    GetNext(T, next);
    while (S[i] != '\0' && T[j] != '\0')
    {
        if (S[i] == T[j]) {i++; j++; }
        else {
             j = next[j];
             if (j == -1) {i++; j++;} }
    }
    if (T[j] == '\0') return (i - j + 1);        //返回本趟匹配的开始位置
    else return 0;
}
```

中国科学院大学
University of Chinese Academy of Sciences

8.2 串匹配问题

- **4. KMP算法**
  - 算法分析
    - 在求得模式 T 的 next 值后，KMP算法只需将主串扫描一遍，设主串的长度为 n，则KMP算法的时间复杂度是O(n)。

作业-课后练习27

- 假设在文本“ababcabccabccacbab”中查找模式“abccac”,分别写出采用BF算法和KMP算法的串匹配过程。

End