第四章 字符串和多维数组

本章的基本内容是：

串的存储结构及模式匹配算法
数组的逻辑结构特征
数组的存储结构及寻址方法
特殊矩阵和稀疏矩阵的压缩存储方法

字符串

字符串的定义

字符串：零个或多个字符组成的有限序列。简称串

串长度：串中所包含的字符个数。

空格串：只包含空格的串。

空串：长度为0的串，记为：" "。

非空串通常记为：

$$S=" s_1\ s_2\ \ldots\ldots\ s_n\ "$$

其中：S是串名，双引号是定界符，双引号引起来的部分是串值，s_i（$1 \leqslant i \leqslant n$）是一个任意字符。

字符串

字符串的定义

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，即6万5千多个字符，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字符集中的前256个字符与扩展ASCII码完全相同。

字符串

字符串的定义

子串：串中任意个连续的字符组成的子序列。

主串：包含子串的串。

子串的位置：子串的第一个字符在主串中的序号。

***S*1="*ab*12*cd* "** //长度为6的串

***S*2= "*ab*12"** //长度为4的串

***S*3= "26138"** //长度为5的串

***S*4="*ab*12ϕ "** //长度为5的串

***S*5= ""** //空串，长度为0

***S*6= " ϕ ϕ ϕ "** //含3个空格的空格串，长度为3

字符串

字符串的定义

串的比较：通过组成串的字符之间的比较来进行的。

给定两个串：$X="x_1x_2\cdots x_n"$和$Y="y_1y_2\cdots y_m"$，则：

1. 当$n=m$且$x_1=y_1$，…，$x_n=y_m$时，称$X=Y$；

2. 当下列条件之一成立时，称$X<Y$：

⑴ $n<m$且$x_i=y_i$（$1\leqslant i\leqslant n$）；

⑵存在$k\leqslant \min(m,n)$，使得$x_i=y_i(1\leqslant i\leqslant k-1)$且$x_k<y_k$。

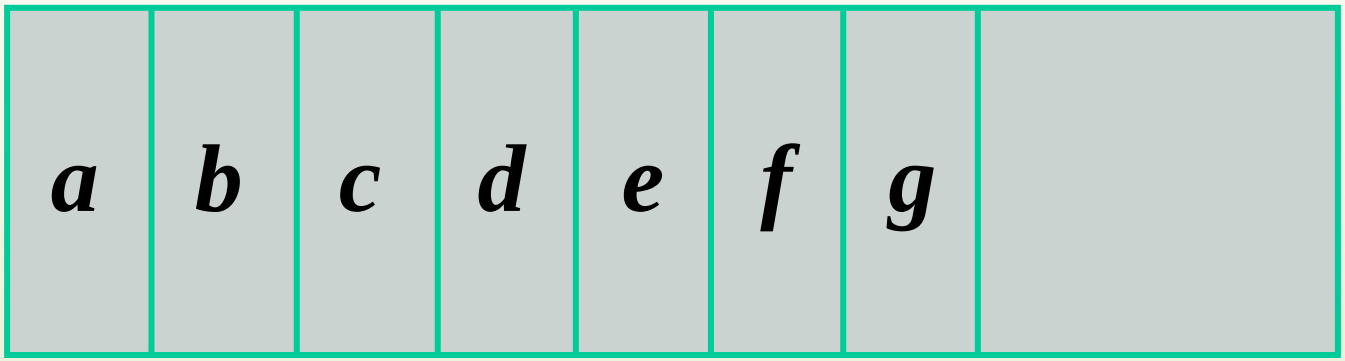
例：$S1="ab12cd"$，$S2="ab12"$，$S3="ab13"$

“串值大小”是按“词典次序”进行比较的。

字符串

串的存储结构

顺序串：用数组来存储串中的字符序列。

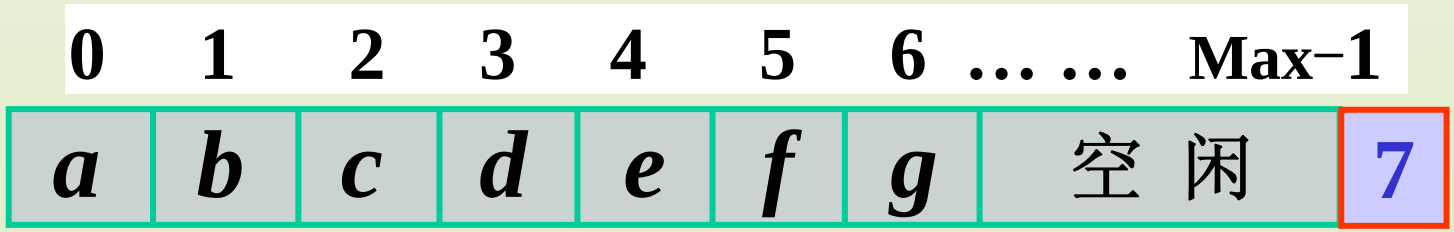


字符串

串的存储结构

如何表示串的长度？

方案1：用一个变量来表示串的实际长度。



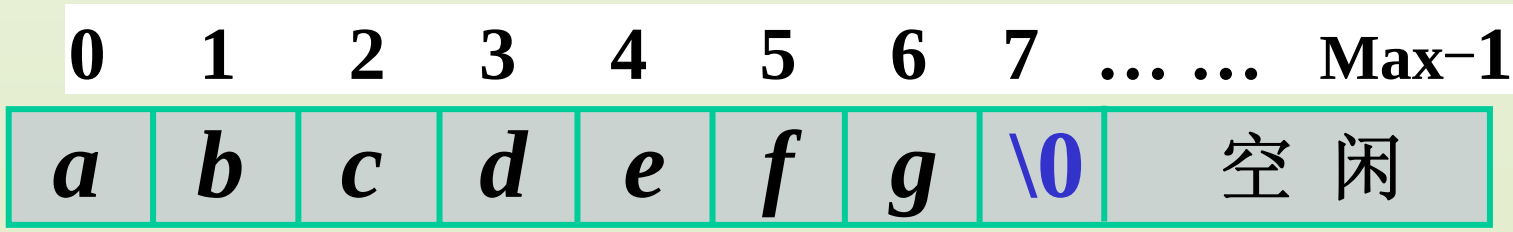
字符串

串的存储结构

如何表示串的长度？

方案1：用一个变量来表示串的实际长度。

方案2：在串尾存储一个不会在串中出现的特殊字符作为串的终结符，表示串的结尾。



字符串

串的存储结构

如何表示串的长度？

方案1：用一个变量来表示串的实际长度。

方案2：在串尾存储一个不会在串中出现的特殊字符作为串的终结符，表示串的结尾。

方案3：用数组的0号单元存放串的长度，从1号单元开始存放串值。

0	1	2	3	4	5	6	7 … …	Max−1
7	*a*	*b*	*c*	*d*	*e*	*f*	*g*	空 闲

字符串

模式匹配

模式匹配：给定主串S="$s_1s_2\ldots s_n$"和模式T="$t_1t_2\ldots t_m$"，在S中寻找T的过程称为模式匹配。如果匹配成功，返回T在S中的位置，如果匹配失败，返回0。

假设串采用顺序存储结构，从数组下标0开始存放字符，在串尾存储 '\0',作为串的终结符。

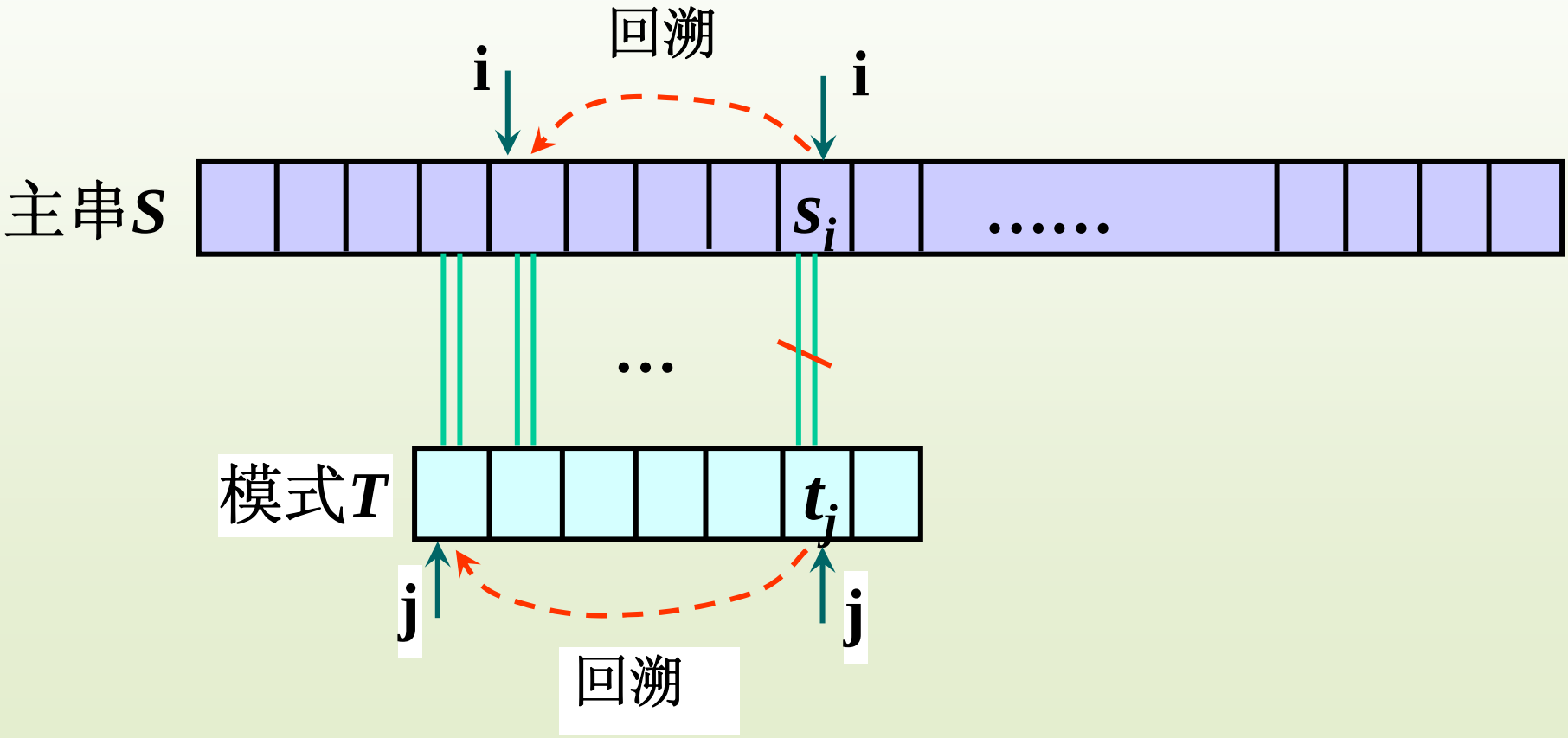
字符串

模式匹配——BF算法 （Brute Force算法）

基本思想：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；否则，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，直到T 中的字符全部比较完毕，则说明本趟匹配成功；或S中字符全部比较完，则说明匹配失败。

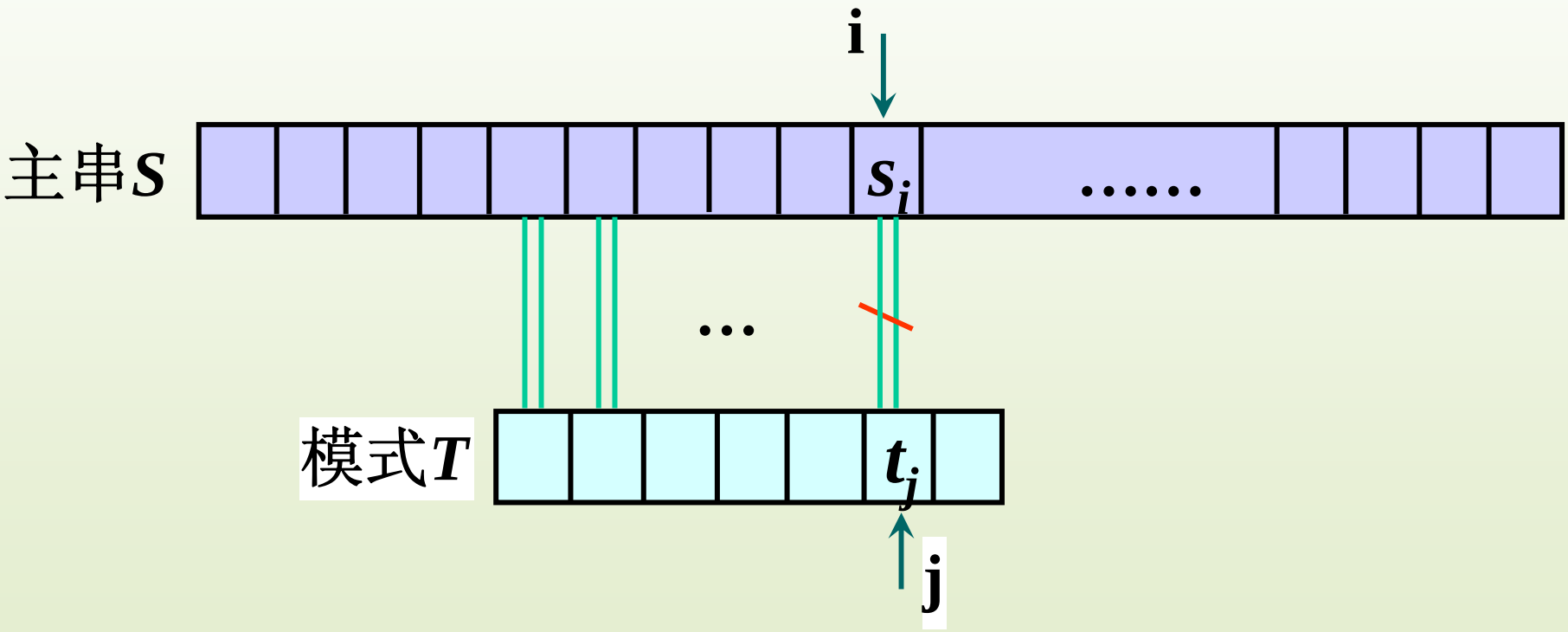
字符串

模式匹配——BF算法



字符串

模式匹配——BF算法

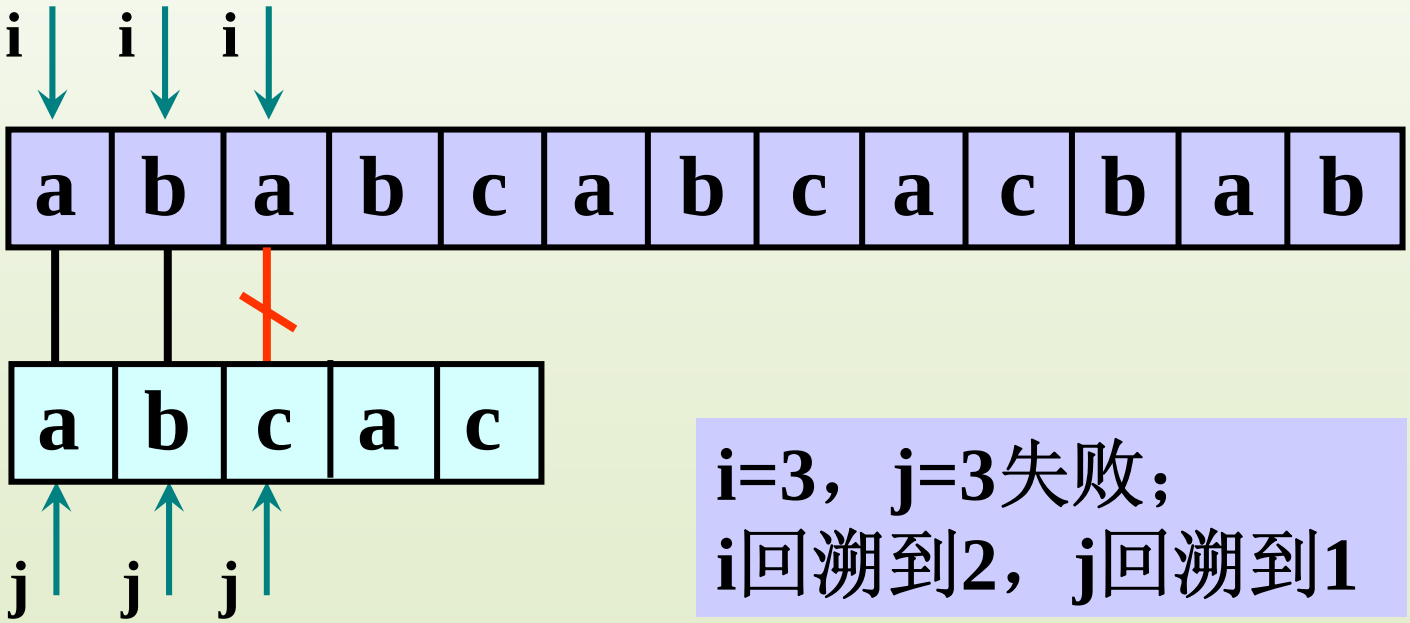


字符串

模式匹配——BF算法

例：主串S="ababcabcacbab"，模式T="abcac"

第1趟



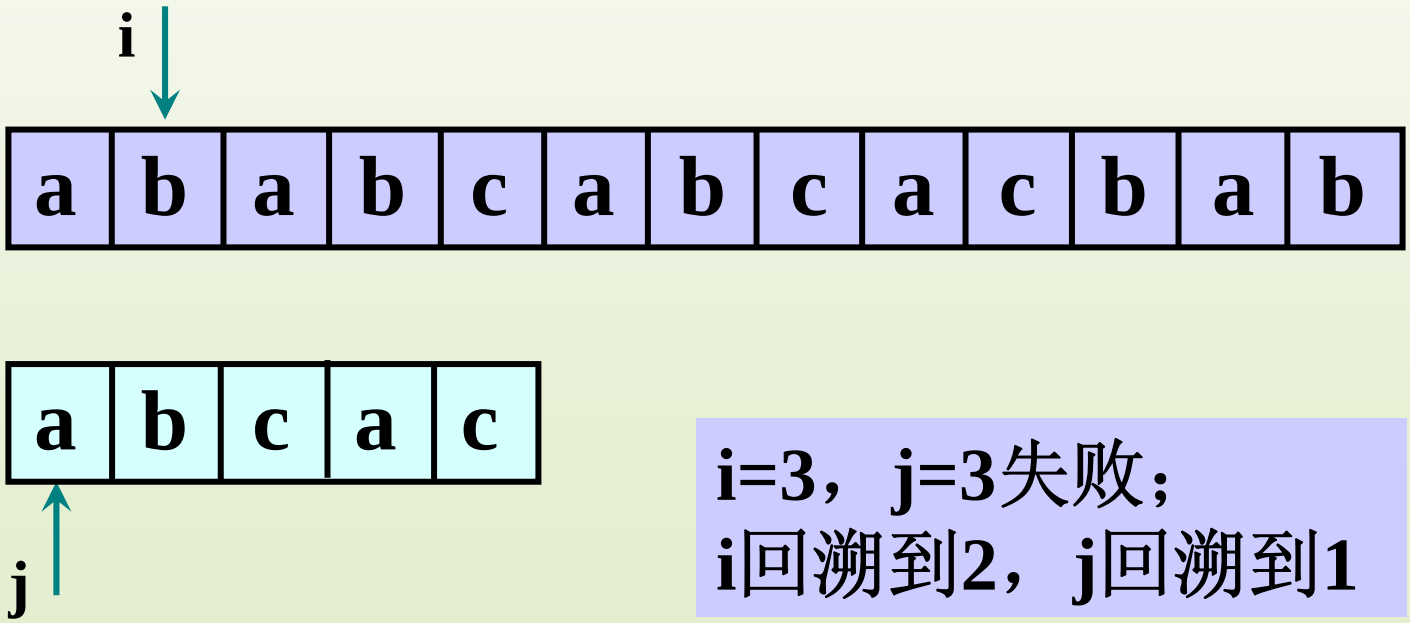
i=3，j=3失败；
i回溯到2，j回溯到1

字符串

模式匹配——BF算法

例：主串S="ababcabcacbab"，模式T="abcac"

第1趟



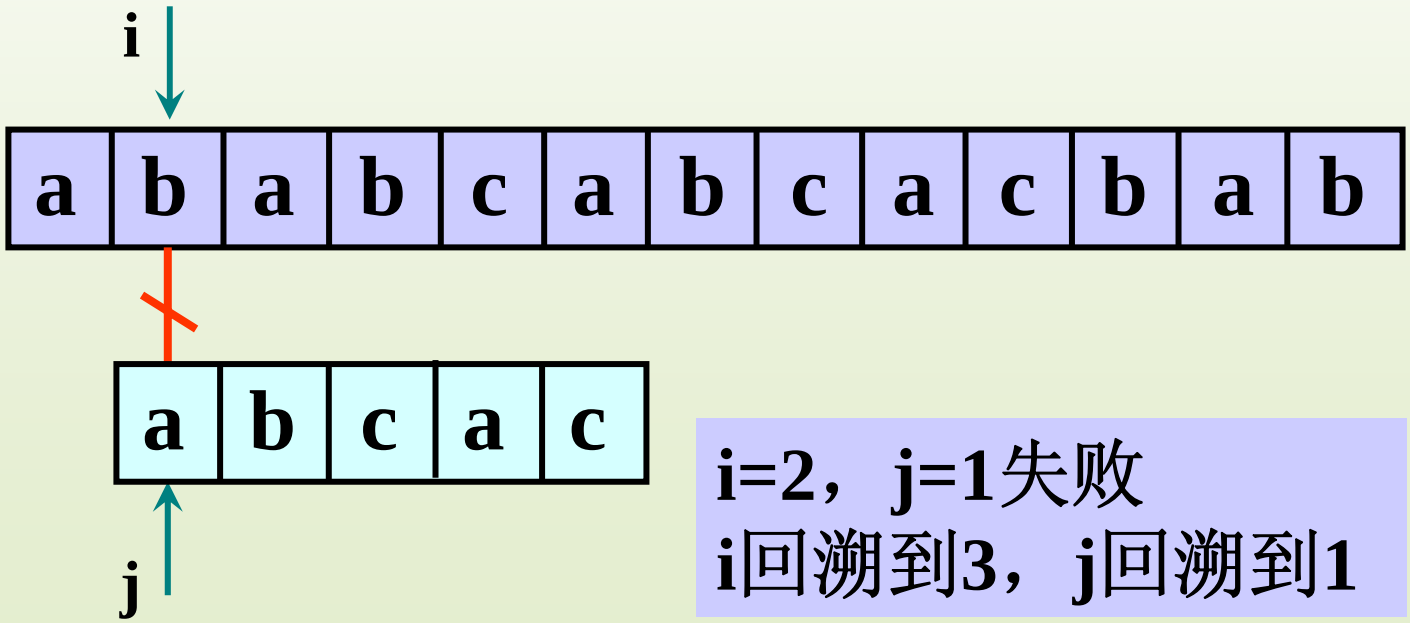
i=3，j=3失败；
i回溯到2，j回溯到1

字符串

模式匹配——BF算法

例：主串S="ababcabcacbab"，模式T="abcac"

第2趟



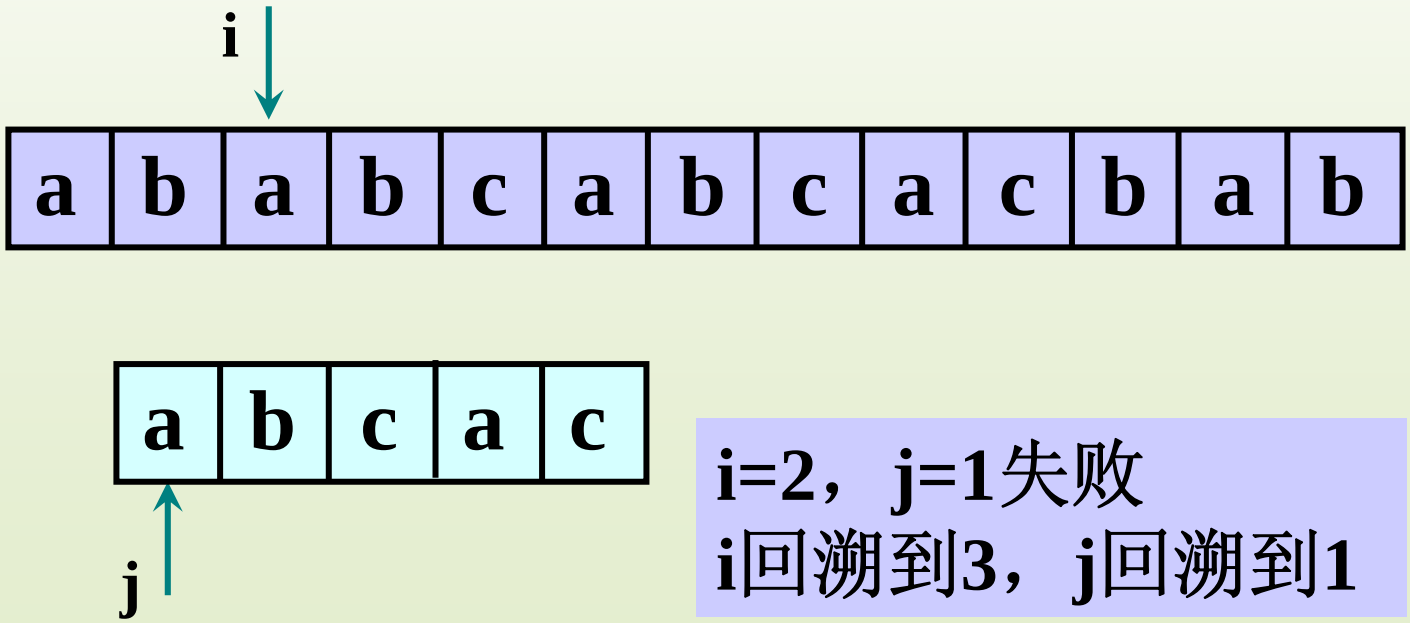
i=2，j=1失败
i回溯到3，j回溯到1

字符串

模式匹配——BF算法

例：主串S="ababcabcacbab"，模式T="abcac"

第2趟

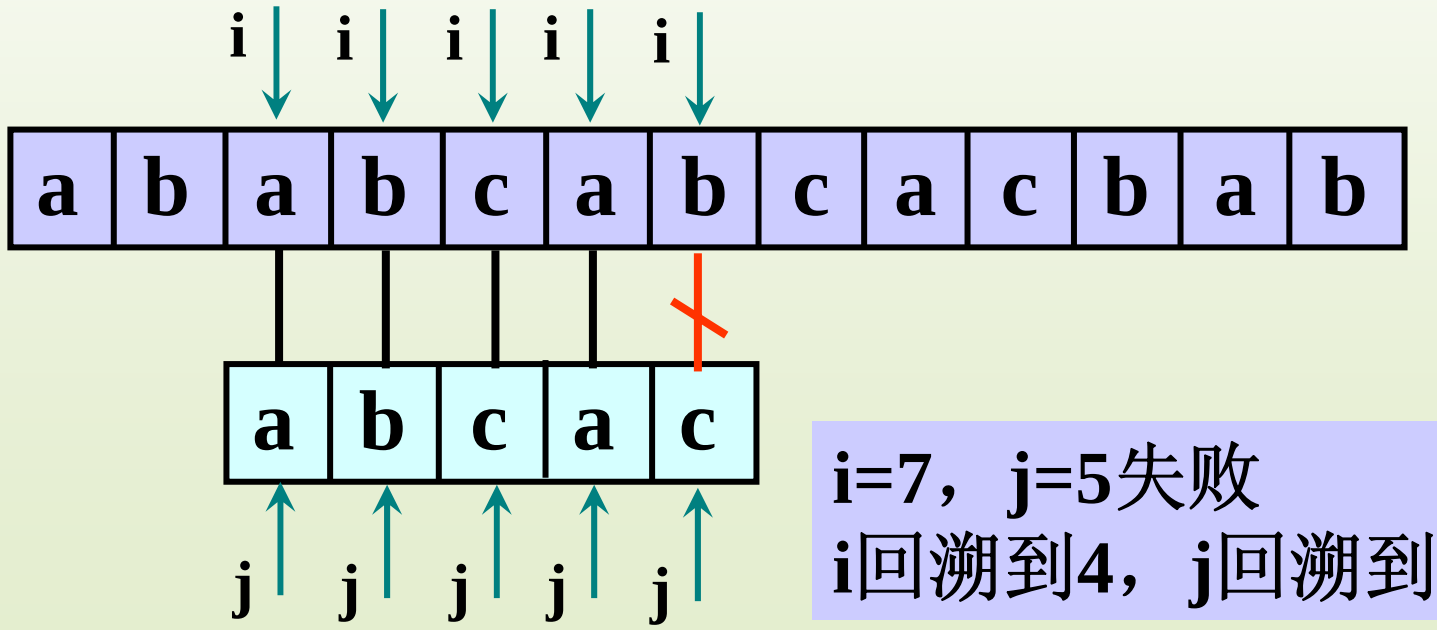


i=2，j=1失败
i回溯到3，j回溯到1

模式匹配——BF算法

例：主串S="ababcabcacbab"，模式T="abcac"

第3趟



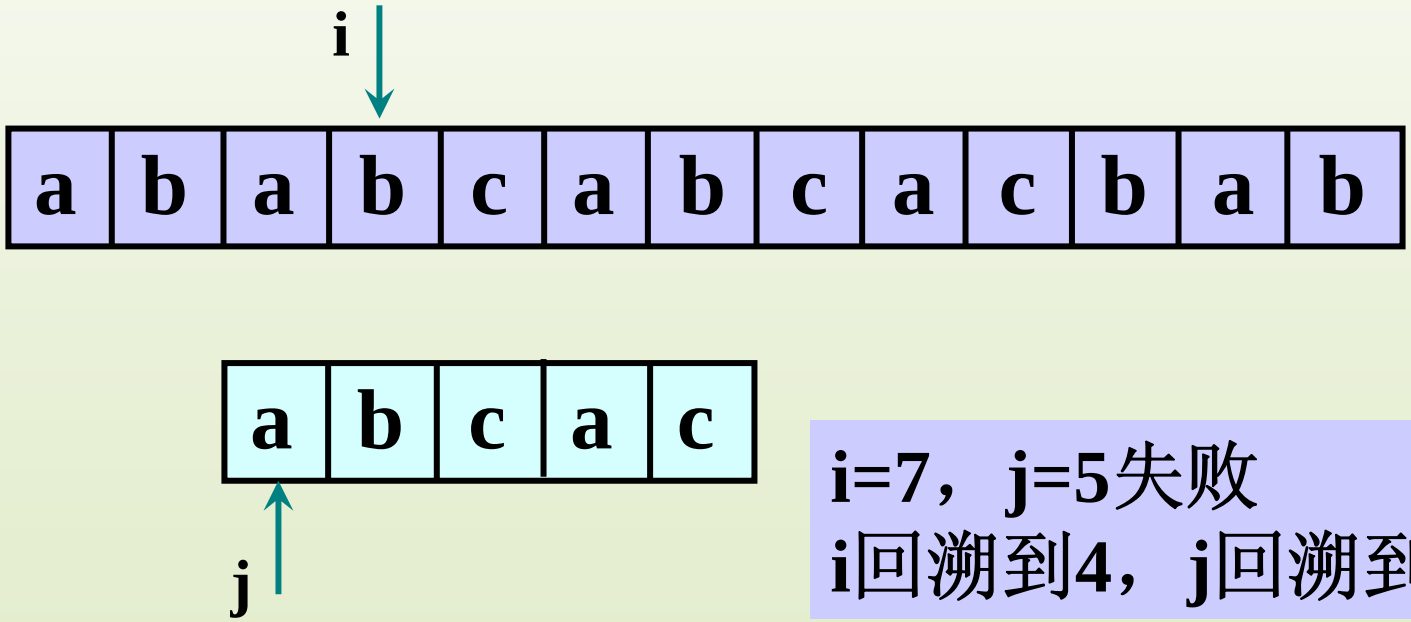
i=7，j=5失败
i回溯到4，j回溯到1

字符串

模式匹配——BF算法

例：主串S="ababcabcacbab"，模式T="abcac"

第3趟



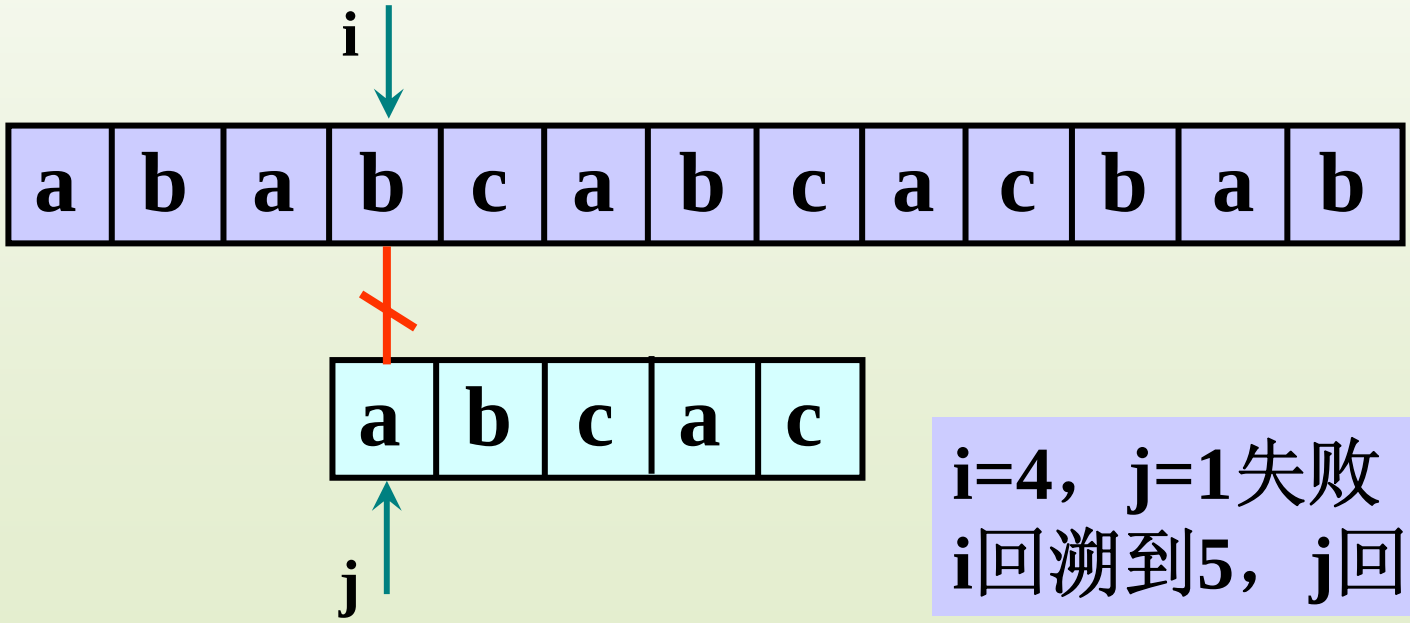
i=7，j=5失败
i回溯到4，j回溯到1

字符串

模式匹配——BF算法

例：主串S="ababcabcacbab"，模式T="abcac"

第4趟



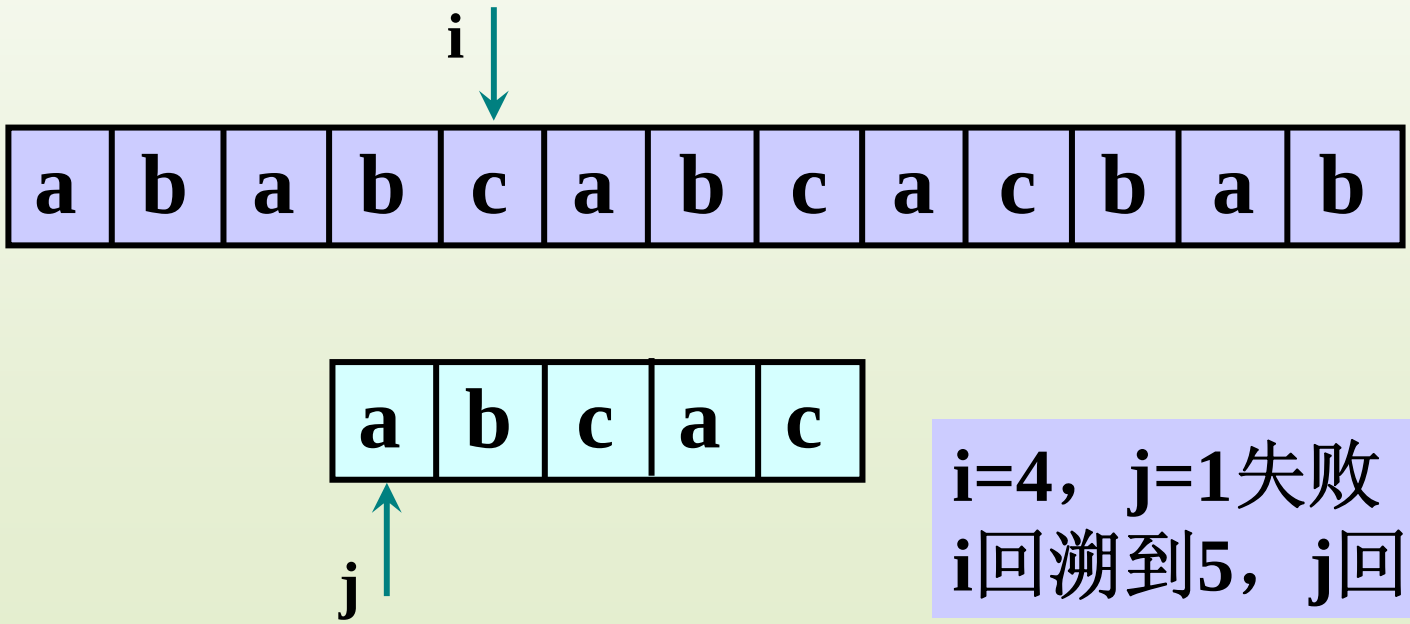
i=4，j=1失败
i回溯到5，j回溯到1

字符串

模式匹配——BF算法

例：主串S="ababcabcacbab"，模式T="abcac"

第4趟



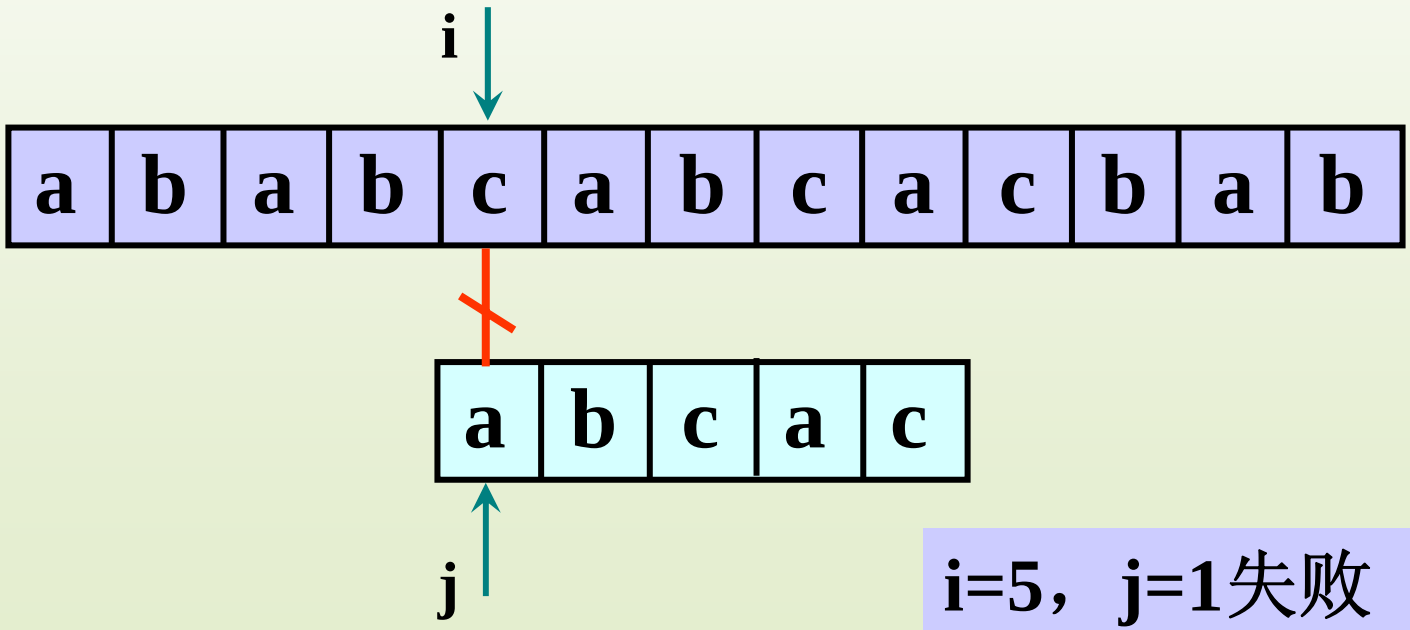
i=4，j=1失败
i回溯到5，j回溯到1

字符串

模式匹配——BF算法

例：主串S="ababcabcacbab"，模式T="abcac"

第5趟



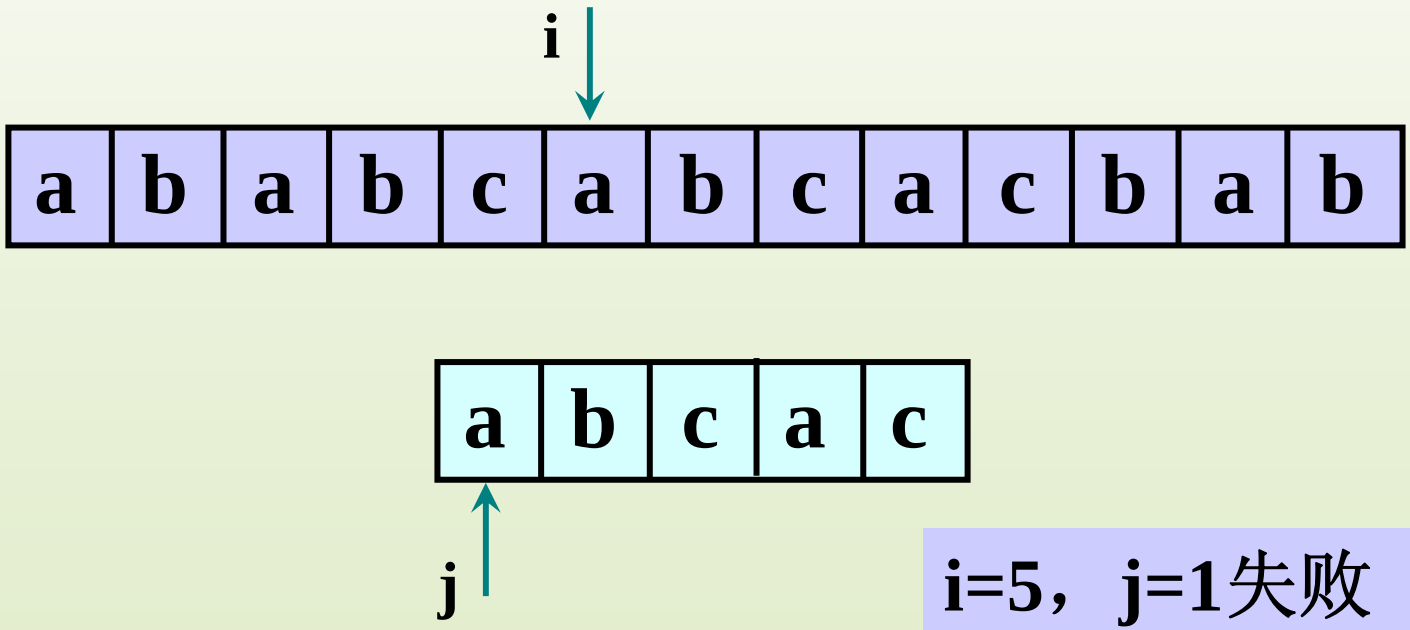
i=5，j=1失败
i回溯到6，j回溯到1

字符串

模式匹配——BF算法

例：主串S="ababcabcacbab"，模式T="abcac"

第5趟



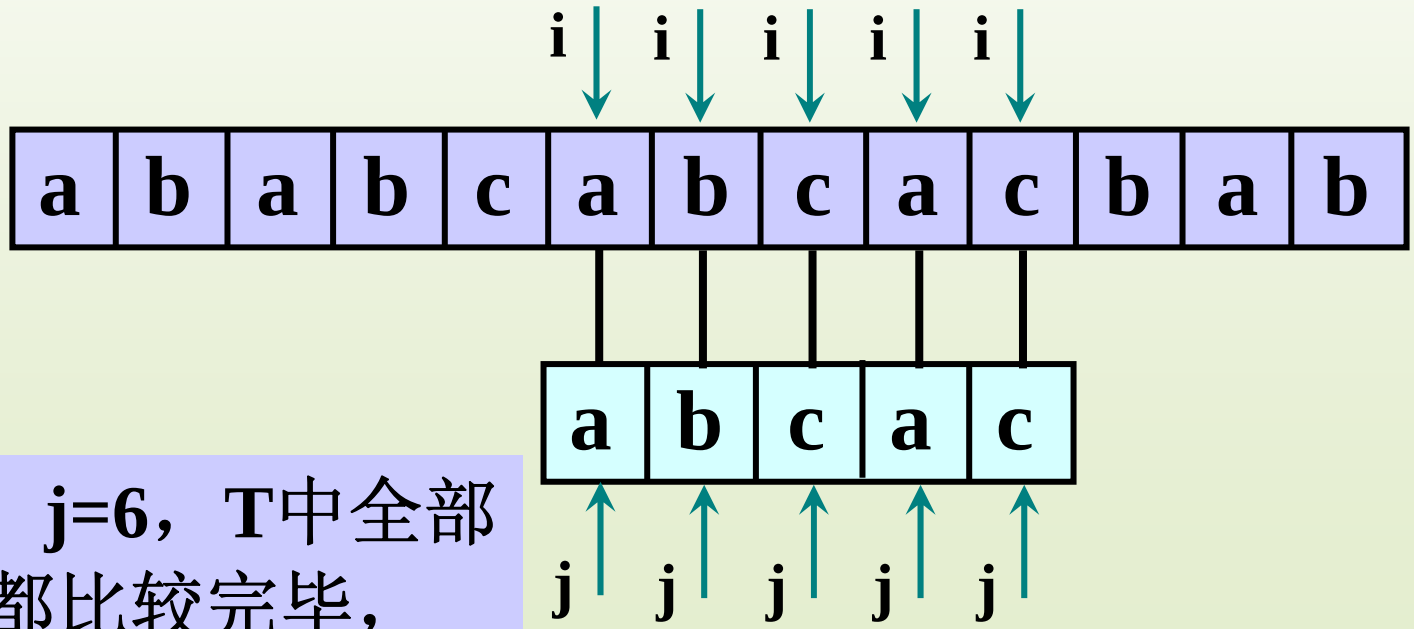
i=5，j=1失败
i回溯到6，j回溯到1

字符串

模式匹配——BF算法

例：主串S="ababcabcacbab"，模式T="abcac"

第6趟



i=11，j=6，T中全部字符都比较完毕，匹配成功。

字符串

模式匹配——BF算法

1. 在串S和串T中设比较的起始下标i和j；
2. 循环直到S或T的所有字符均比较完；
   2.1 如果S[i]=T[j]，继续比较S和T的下一个字符；
   2.2 否则，将i和j回溯，准备下一趟比较；
3. 如果T中所有字符均比较完，则匹配成功，返回匹配的起始比较下标；否则，匹配失败，返回0；

字符串

模式匹配——BF算法

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

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

字符串

模式匹配——BF算法

设串***S***长度为***n***，串***T***长度为***m***，在匹配成功的情况下，考虑两种极端情况：

(1) 最好：不成功的匹配都发生在串***T***的第一个字符。

例如：***S="aaaaaaaaaabcdccccc"***

T="bcd "

字符串

模式匹配——BF算法

设串S长度为n，串T长度为m，在匹配成功的情况下，考虑两种极端情况：

最好情况：不成功的匹配都发生在串T的第1个字符。

设匹配成功发生在s_i处，则在i-1趟不成功的匹配中共比较了i-1次，第i趟成功的匹配共比较了m次，所以总共比较了i-1+m次，所有匹配成功的可能情况共有n-m+1种（s_i可能取值：$1,2,\ldots,n-m+1$），则：

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

字符串

模式匹配——BF算法

设串*S*长度为*n*，串*T*长度为*m*，在匹配成功的情况下，考虑两种极端情况：

最坏情况：不成功的匹配都发生在串*T*的最后一个字符。

例如：*S*="*aaaaaaaaaaaabccccc*"

T="*aaab*"

字符串

模式匹配——BF算法

设串***S***长度为***n***，串***T***长度为***m***，在匹配成功的情况下，考虑两种极端情况：

最坏情况：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，则在***i*-1**趟不成功的匹配中共比较了**(*i*-1)**×***m***次，第***i***趟成功的匹配共比较了***m***次，所以总共比较了***i***×***m***次，因此

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

字符串

模式匹配——KMP算法

KMP算法是模式匹配算法的一种改进算法，是D.E.Knuth与J.H.Morris和V.R.Pratt 同时发现的，因此人们称它为克努特-莫里斯-普拉特操作（简称KMP算法）。

为什么**BF**算法时间性能低？

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。

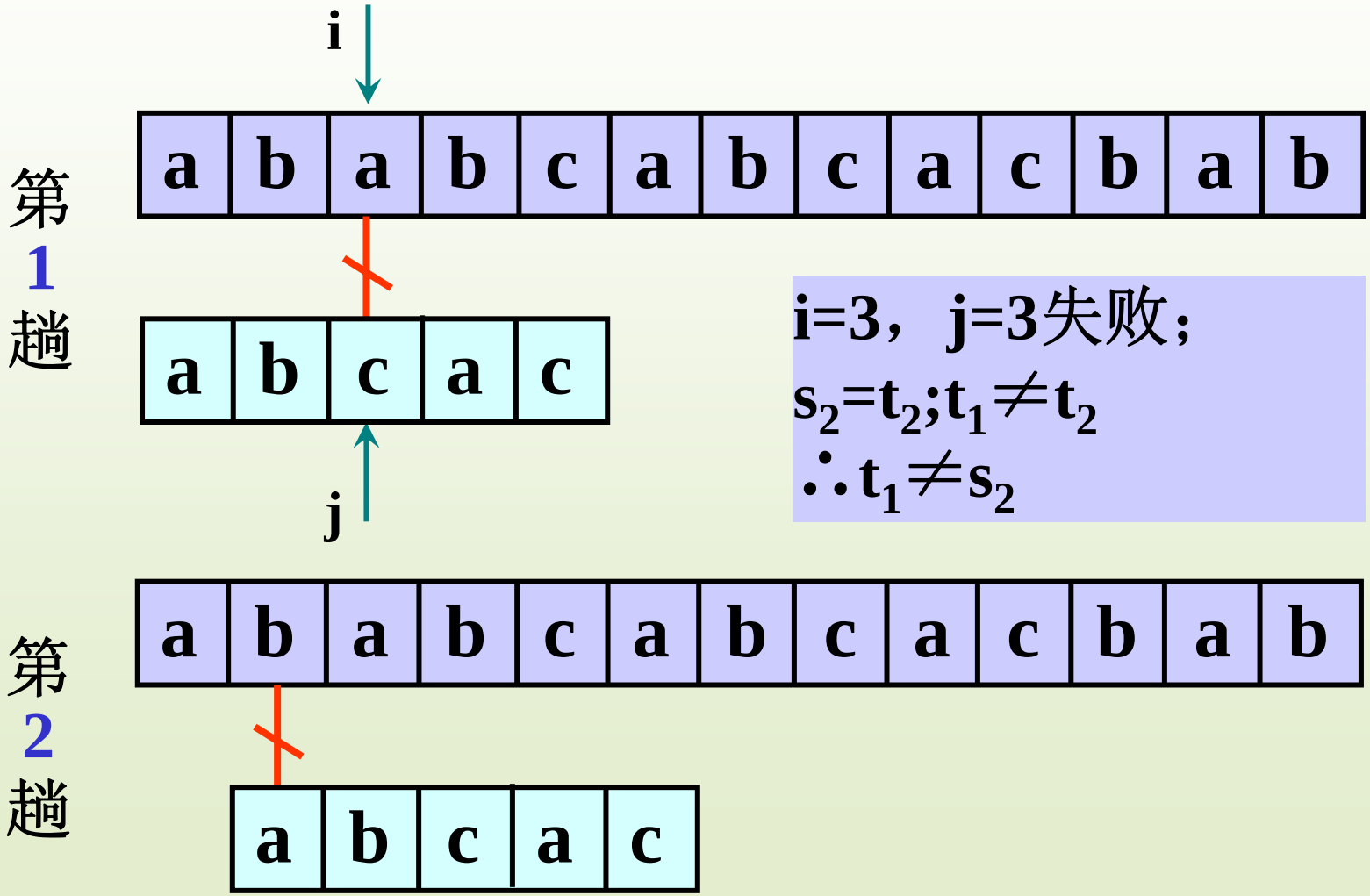
如何在匹配不成功时主串不回溯？

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？

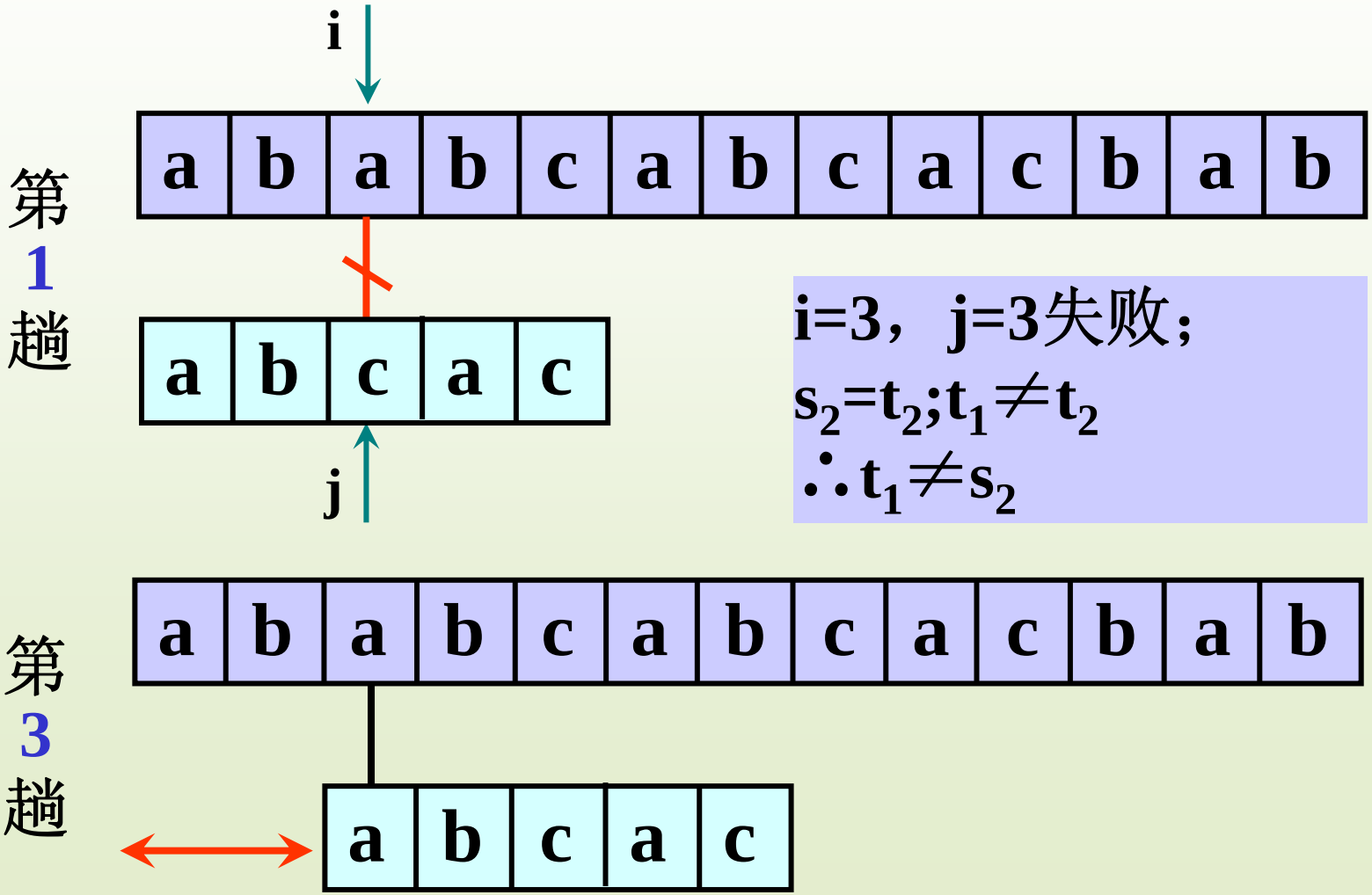
字符串

模式匹配——KMP算法



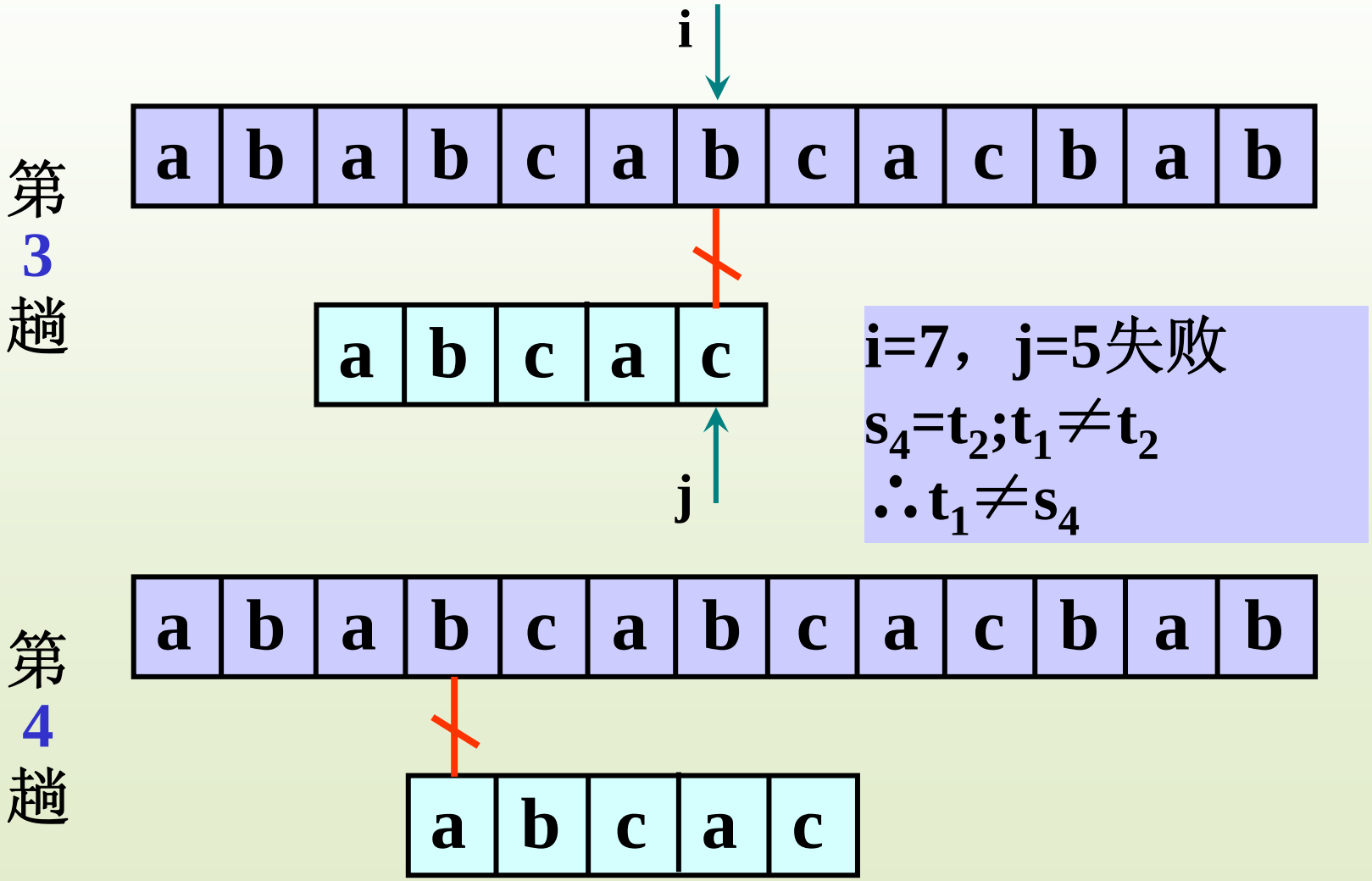
字符串

模式匹配——KMP算法



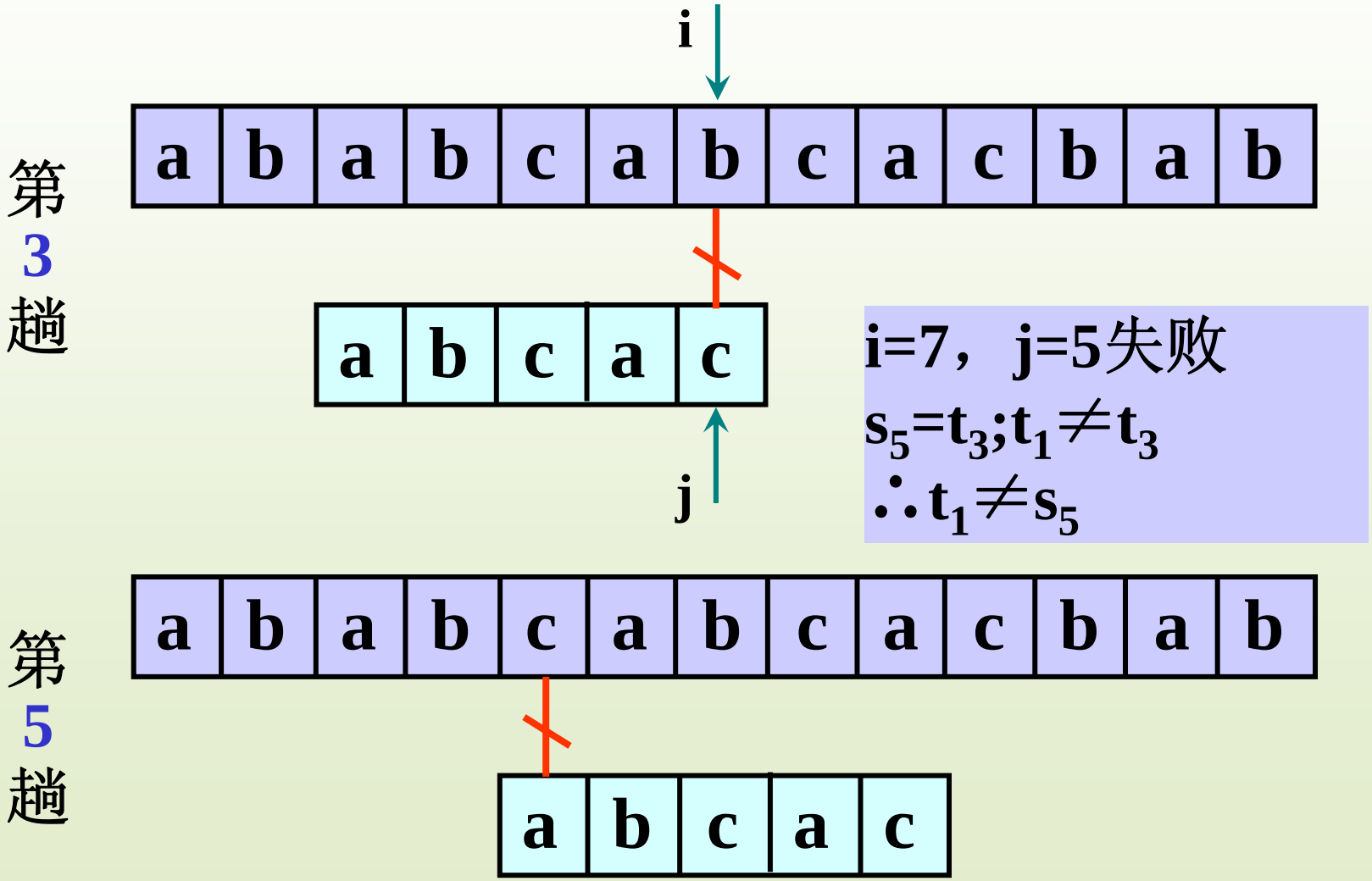
字符串

模式匹配——KMP算法



字符串

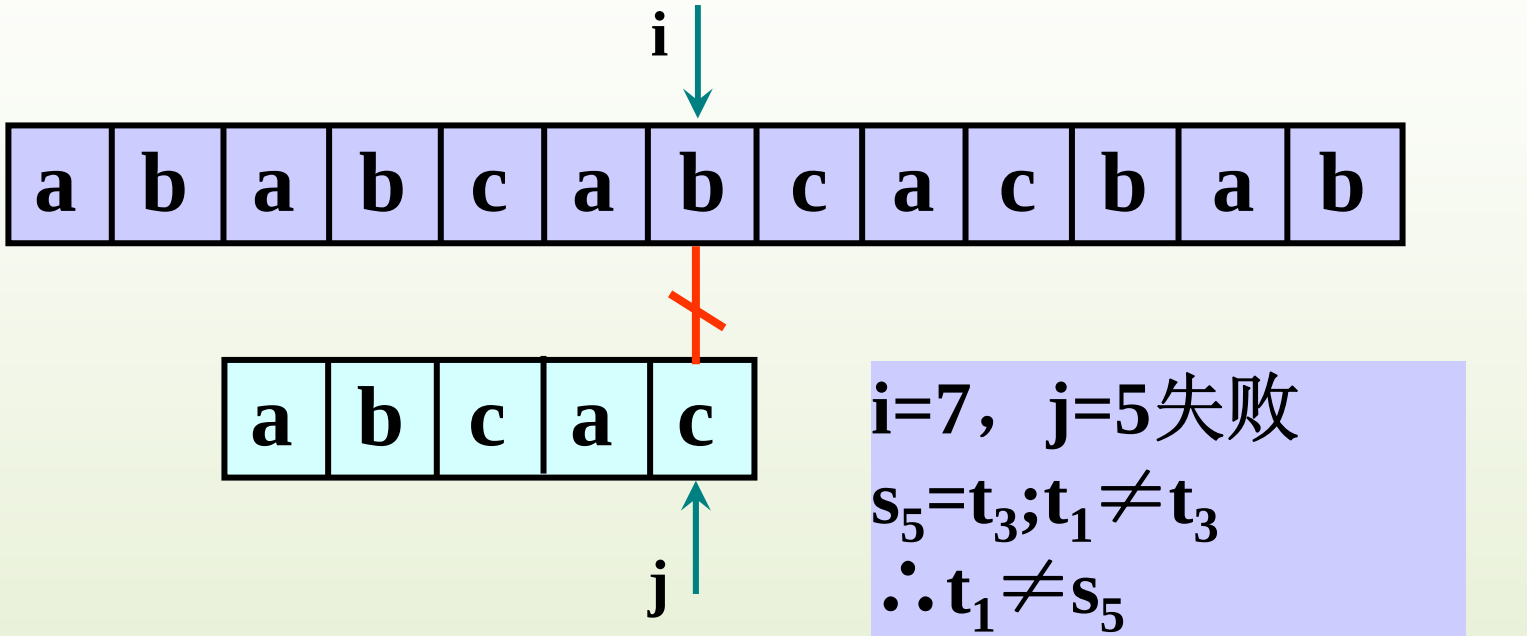
模式匹配——KMP算法



字符串

模式匹配——KMP算法

第3趟

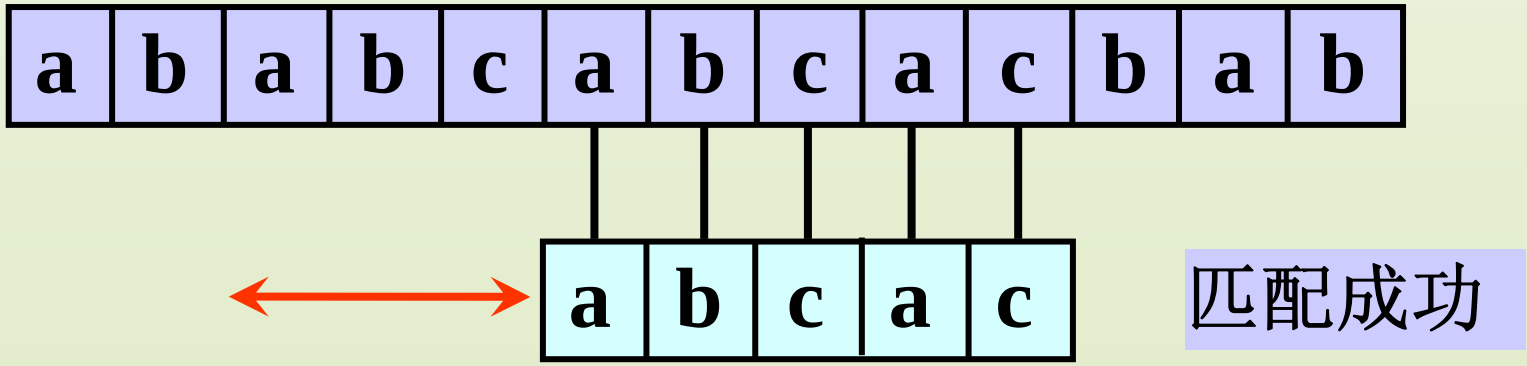


i=7，j=5失败

$s_5=t_3;t_1\neq t_3$

$\therefore t_1\neq s_5$

第6趟



匹配成功

字符串

模式匹配——KMP算法

结论： i可以不回溯，模式向右滑动到的新比较起点k ，并且k 仅与模式串T有关！

需要讨论两个问题:

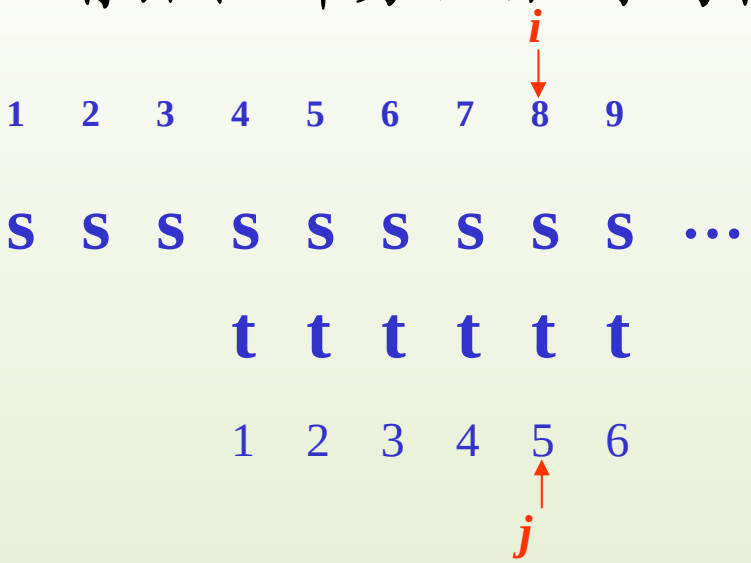
①如何由当前部分匹配结果确定模式向右滑动的新比较起点k?

②模式应该向右滑多远才是最高效率的?

字符串

模式匹配——KMP算法

请抓住部分匹配时的特征:



设 i=8,j=5时失配，有$s_4s_5s_6s_7=t_1t_2t_3t_4$, ①

模式滑动到第k个字符：k＝

1:一般情况

2：有$s_7=t_1$，②，由①和 ②有$t_1=t_4$

3：有$s_6s_7=t_1t_2$，③，由①和 ③有$t_1t_2=t_3t_4$

4：有$s_5s_6s_7=t_1t_2t_3$，④，由①和 ④ 有$t_1t_2t_3=t_2t_3t_4$

结论:

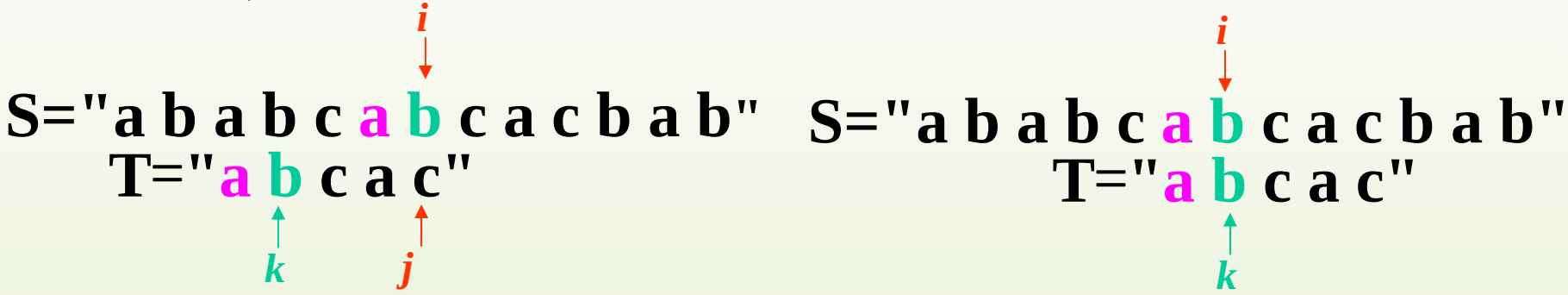
（1）滑动位置*k*仅与模式串*T*有关;

（2） *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*（即比较的新起点）。

字符串

模式匹配——KMP算法

请抓住部分匹配时的两个特征:

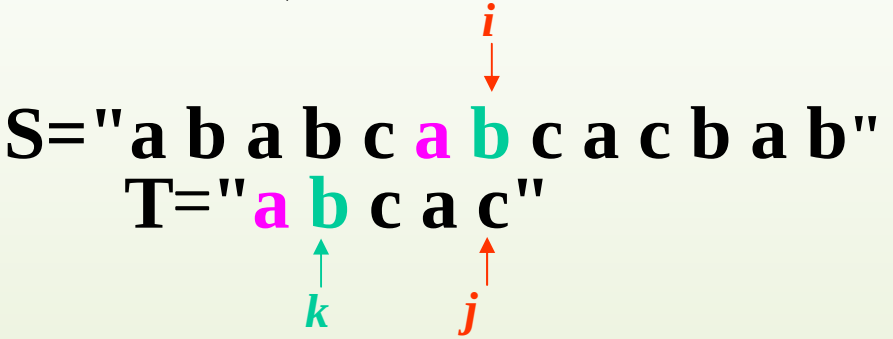


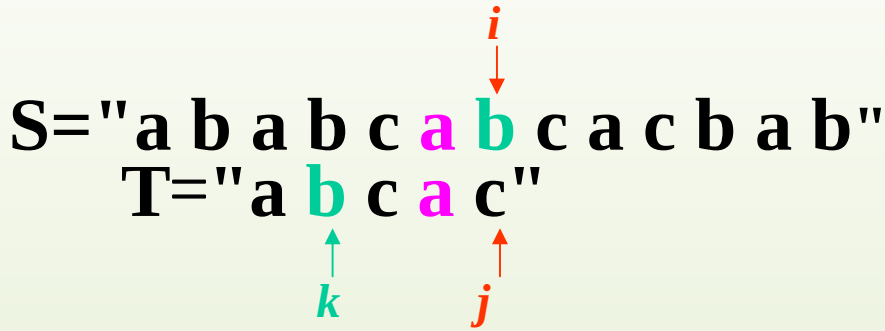
（1）设模式滑动到第k个字符，则$T_1 \sim T_{k-1} = S_{i-(k-1)} \sim S_{i-1}$

字符串

模式匹配——KMP算法

请抓住部分匹配时的两个特征:





（1）设模式滑动到第k个字符，则$T_1 \sim T_{k-1} = S_{i-(k-1)} \sim S_{i-1}$

（2）则$T_{j-(k-1)} \sim T_{j-1} = S_{i-(k-1)} \sim S_{i-1}$

两式联立可得: $T_1 \sim T_{k-1} = T_{j-(k-1)} \sim T_{j-1}$

字符串

模式匹配——KMP算法

$T_1 \cdots T_{k-1} = T_{j-(k-1)} \cdots T_{j-1}$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（即比较的新起点）；

（2）滑动位置k 仅与模式串T有关。

$T_1 \cdots T_{k-1} = T_{j-(k-1)} \cdots T_{j-1}$的物理意义是什么？

从第1位往右经过k-1位

从j-1位往左经过k-1位

模式应该向右滑多远才是最高效率的?

$$k = \max\{\, k \mid 1<k<j \text{ 且} T_1 \cdots T_{k-1} = T_{j-(k-1)} \cdots T_{j-1} \,\}$$

字符串

模式匹配——KMP算法

令k = next[j]，则：

$$next[j]=\begin{cases}0 & \text{当}j=1\text{时} \quad \text{//不比较}\\ \max\{k \mid 1<k<j \ \text{且} \ T_1\cdots T_{k-1}=T_{j-(k-1)}\cdots T_{j-1}\} & \\ 1 & \text{其他情况}\end{cases}$$

next[j]函数表征着模式T中最大相同首子串和尾子串（真子串）的长度。

可见，模式中相似部分越多，则next[j]函数越大，它既表示模式 T 字符之间的相关度越高，模式串向右滑动得越远，与主串进行比较的次数越少，时间复杂度就越低。

模式匹配——KMP算法

（2）计算next[j]的方法：

- 当j=1时，next[j]=0;

//next[j]=0表示根本不进行字符比较。(i++，j=1)

- 当j>1时，next[j]的值为：模式串的位置从1到j-1构成的串中所出现的首尾相同的子串的最大长度加1。
- 当无首尾相同的子串时next[j]的值为1。

//next[j]=1表示从模式串头部开始进行字符比较

字符串

模式匹配——KMP算法

模式串 T:	a	b	a	a	b	c	a	c
可能失配位 j:	1	2	3	4	5	6	7	8
新匹配位k=next[j] :	0	1	1	2	2	3	1	2

j=1时, next[j] = 0;

j=2时, next[j] = 1;

j=3时, $T_1 \neq T_2$，因此，k=1;

j=4时, $T_1 = T_3$，因此，k=2;

j=5时, $T_1 = T_4$，因此，k=2;

以此类推。

作业：

1. 已知：

S="a b a b c a b a a b c a a b a a b a b c a a b"

T="a b a a b a b c"

求模式**T**的**next[j]**（**01122343**），写出**KMP**匹配过程。

2. 设输入元素为**1,2,3,a,b**，输入次序为**123ab**，元素经过栈后到达输出序列，当所有元素均到达输出序列后，有哪些序列可作为高级语言变量名。

字符串

KMP算法用伪代码描述

1. 在串S和串T中分别设比较的起始下标i和j；
2. 循环直到S中所剩字符长度小于T的长度或T中所有字符均比较完毕

   2.1 如果S[i]=T[j]，继续比较S和T的下一个字符；否则

   2.2 将j向右滑动到next[j]位置，即j=next[j]；

   2.3 如果j=0，则将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，则返回匹配的起始下标；否则返回0；

字符串

求模式串T的next函数值算法

```
void GetNext（char T[ ], int next[ ]）
{
   next[1]=0;     j=1;  k=0;
    while（j<T[0]）
    if（（k==0）||（T[j]= =T[k]））{
          j++;
          k++;
          next[j]=k;
    }
    else k=next[k];
}
```

多维数组

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

特殊线性表

栈——仅在表尾进行插入和删除操作的线性表。

队列——在一端进行插入操作，而另一端进行删除操作的线性表。

串——零个或多个字符组成的有限序列 。

限制元素类型为字符

线性表——具有相同类型的数据元素的有限序列。

多维数组

线性表——具有相同类型的数据元素的有限序列。

将元素的类型进行扩充

（多维）数组——线性表中的数据元素可以是线性表，但所有元素的类型相同。

多维数组

数组的定义

数组是由一组类型相同的数据元素构成的有序集合，每个数据元素称为一个数组元素（简称为元素），每个元素受$n(n \geq 1)$个线性关系的约束，每个元素在n个线性关系中的序号i_1、i_2、…、i_n称为该元素的下标，并称该数组为 n 维数组。

数组的特点

- 元素本身可以具有某种结构，属于同一数据类型；
- 数组是一个具有固定格式和数量的数据集合。

多维数组

数组示例

$$A=\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$

例如，元素a_{22}受两个线性关系的约束，在行上有一个行前驱a_{21}和一个行后继a_{23}，在列上有一个列前驱a_{12}和和一个列后继a_{32}。

多维数组

数组——线性表的推广

$$A=\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$

$A=(A_1, A_2, \ldots\ldots, A_n)$

其中：

$A_i=(a_{1i}, a_{2i}, \ldots\ldots, a_{mi})$

$(1 \leqslant i \leqslant n)$

二维数组是数据元素为线性表的线性表。

多维数组

数组的基本操作

在数组中插入（或）一个元素有意义吗？

将元素 *x* 插入
到数组中第1行第2列。

$$A=\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$

删除数组中
第1行第2列元素。

$$A=\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$

多维数组

数组的基本操作

(1) 存取：给定一组下标，读出对应的数组元素；

(2) 修改：给定一组下标，存储或修改与其相对应的数组元素。

存取和修改操作本质上只对应一种操作——寻址

数组应该采用何种方式存储？

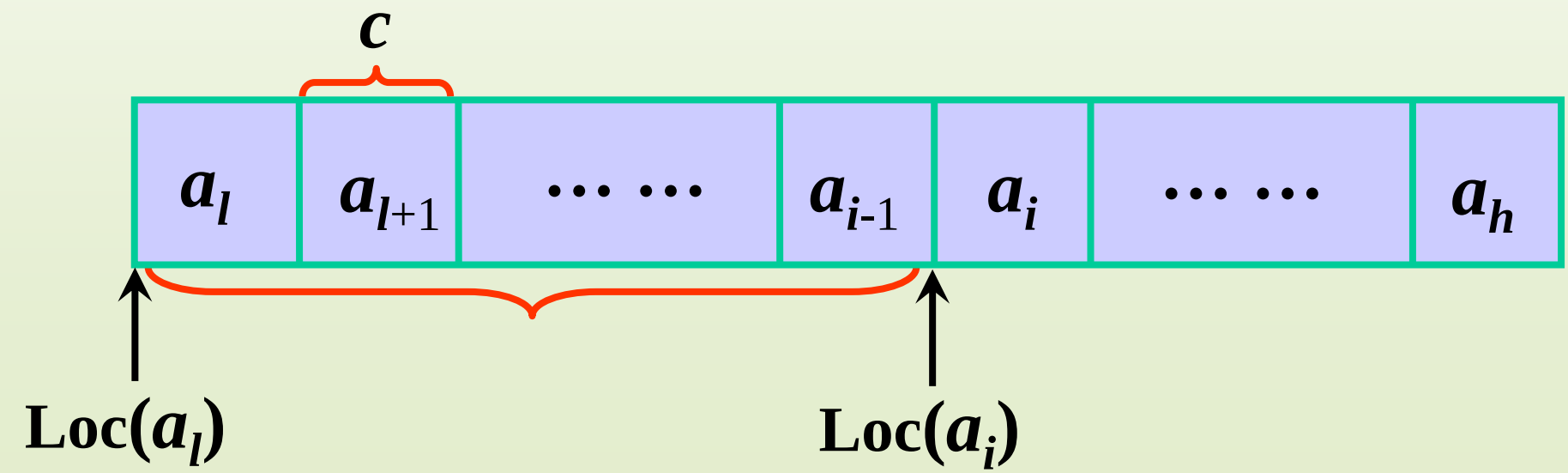
数组没有插入和删除操作，所以，不用预留空间，适合采用顺序存储。

多维数组

数组的存储结构与寻址——一维数组

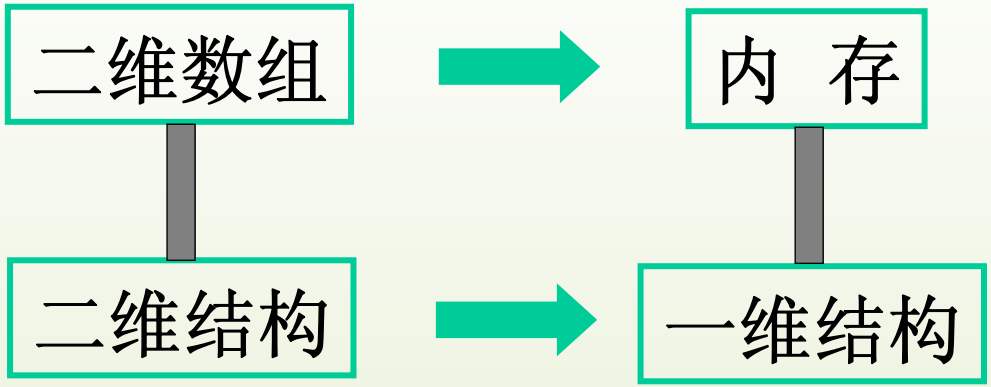
设一维数组的下标的范围为闭区间［l，h］，每个数组元素占用 c 个存储单元，则其任一元素 a_i 的存储地址可由下式确定：

$$\mathrm{Loc}(a_i)=\mathrm{Loc}(a_l)+(i-l)\times c$$



多维数组

数组的存储结构与寻址——二维数组

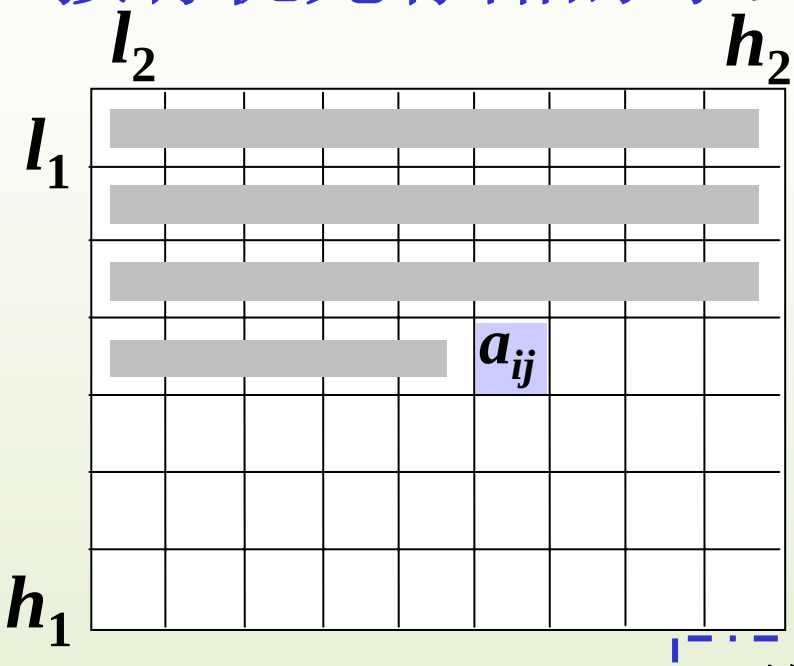


常用的映射方法有两种：

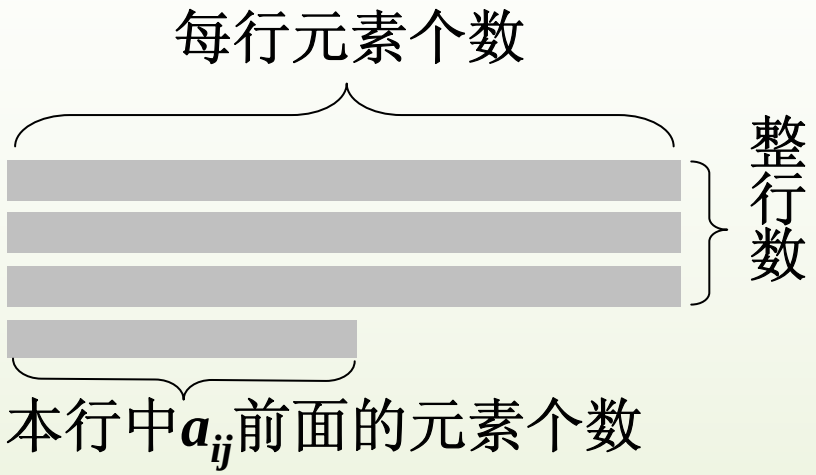
- 按行优先：先行后列，先存储行号较小的元素，行号相同者先存储列号较小的元素。
- 按列优先：先列后行，先存储列号较小的元素，列号相同者先存储行号较小的元素。

多维数组

按行优先存储的寻址



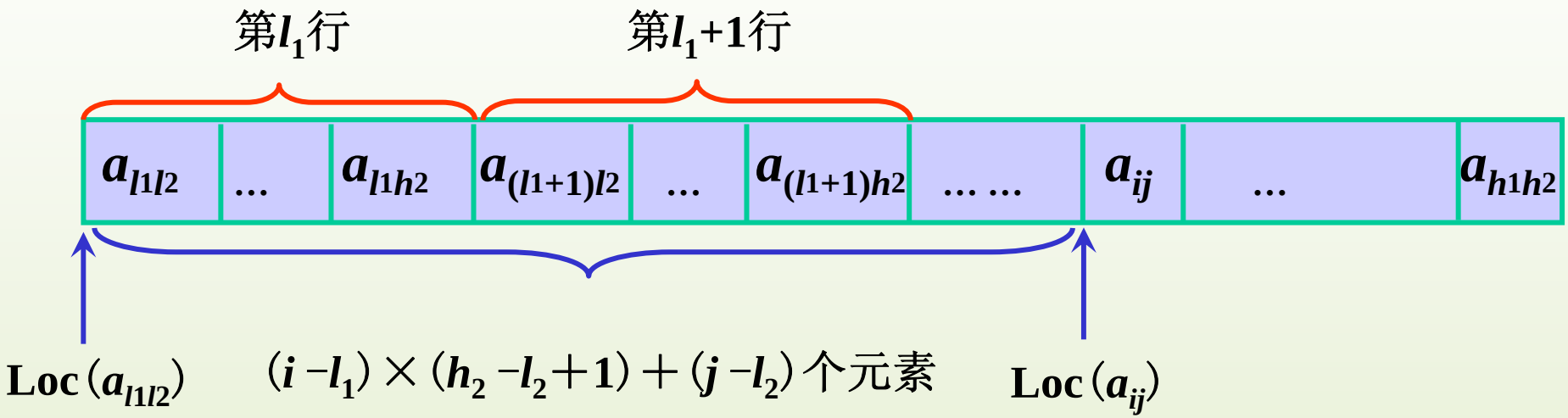
(a) 二维数组



a_{ij}前面的元素个数
=阴影部分的面积
=整行数×每行元素个数+本行中a_{ij}前面的元素个数
$=(i-l_1)\times(h_2-l_2+1)+(j-l_2)$

多维数组

按行优先存储的寻址



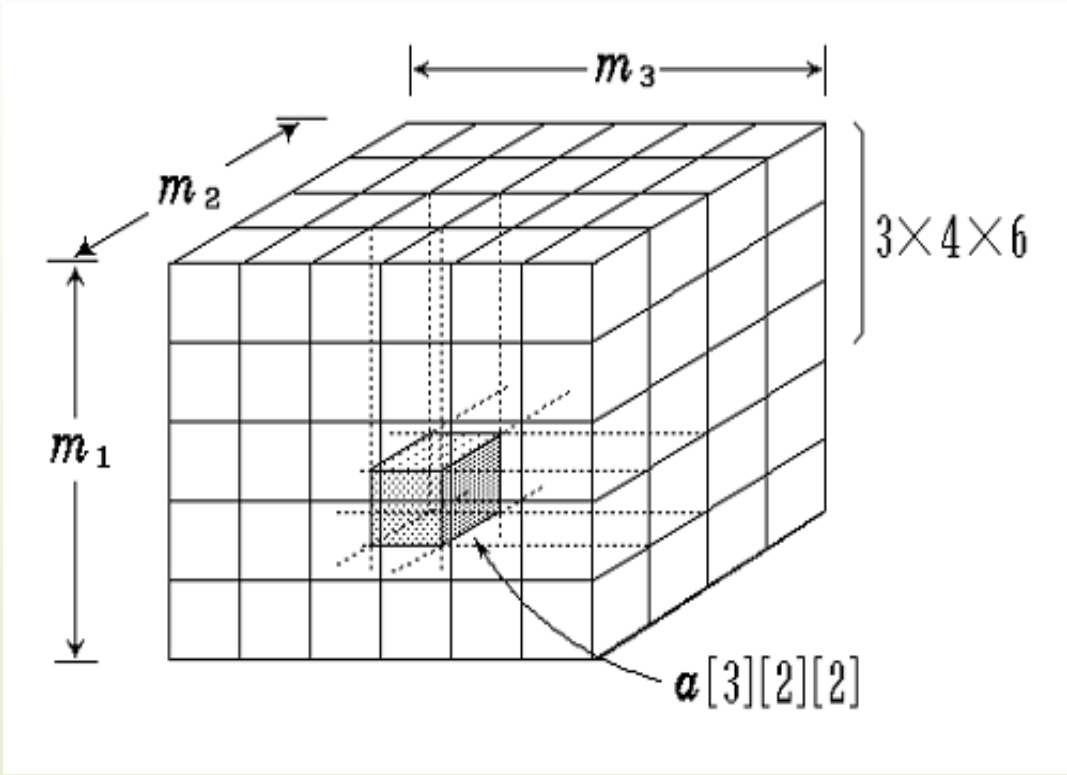
$$\mathrm{Loc}(a_{ij})=\mathrm{Loc}(a_{l1l2})+((i-l1)\times(h2-l2+1)+(j-l2))\times c$$

按列优先存储的寻址方法与此类似。

多维数组

数组的存储结构与寻址——多维数组

n（$n>2$）维数组一般也采用按行优先和按列优先两种存储方法。任一元素存储地址的计算方法



$$\mathrm{Loc}(a_{ijk}) = \mathrm{Loc}(a_{000}) + (i \times m_2 \times m_3 + j \times m_3 + k) \times c$$

矩阵的压缩存储

特殊矩阵和稀疏矩阵

特殊矩阵：矩阵中很多值相同的元素并且它们的分布有一定的规律。

稀疏矩阵：矩阵中有很多零元素。

压缩存储的基本思想是：

(1) 为多个值相同的元素只分配一个存储空间；

(2) 对零元素不分配存储空间。

矩阵的压缩存储

特殊矩阵的压缩存储——对称矩阵

$$A=\begin{pmatrix} 3 & 6 & 4 & 7 & 8 \\ 6 & 2 & 8 & 4 & 2 \\ 4 & 8 & 1 & 6 & 9 \\ 7 & 4 & 6 & 0 & 5 \\ 8 & 2 & 9 & 5 & 7 \end{pmatrix}$$

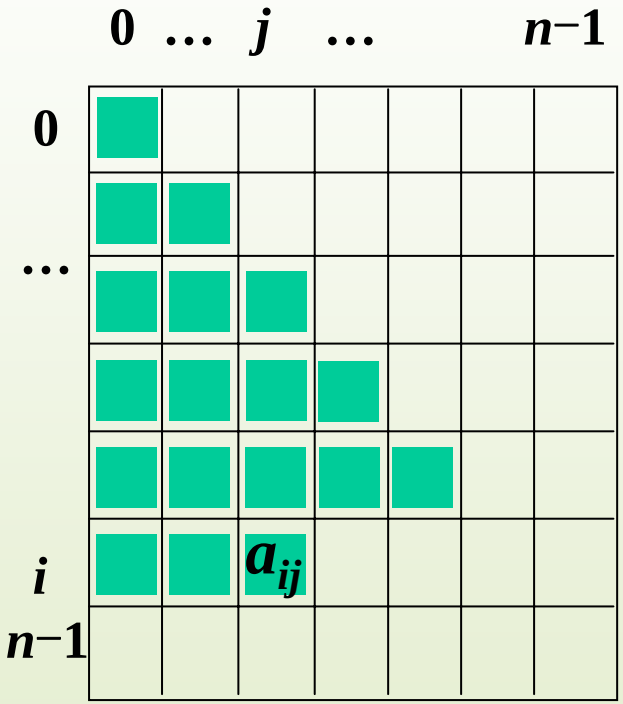
对称矩阵特点：$a_{ij}=a_{ji}$

如何压缩存储？

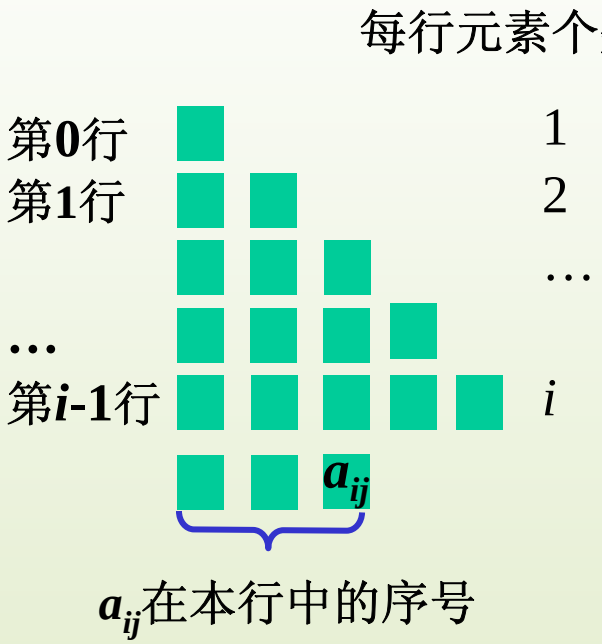
只存储下三角部分的元素。

矩阵的压缩存储

对称矩阵的压缩存储



(a) 下三角矩阵



(b) 存储说明

a_{ij}在一维数组中的序号

$=(1+2+\ldots+i)+j+1$

$= i\times(i+1)/2+ j+1$

∵一维数组下标从0开始

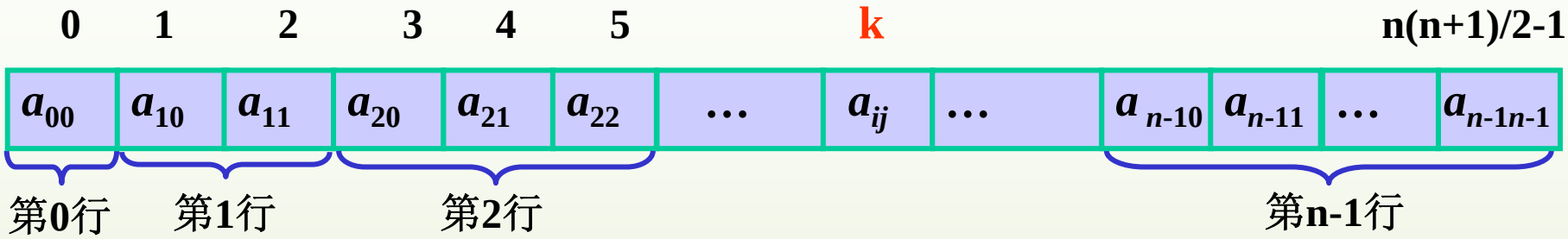
∴a_{ij}在一维数组中的下标

$k= i\times(i+1)/2+ j$

(c) 计算方法

矩阵的压缩存储

对称矩阵的压缩存储

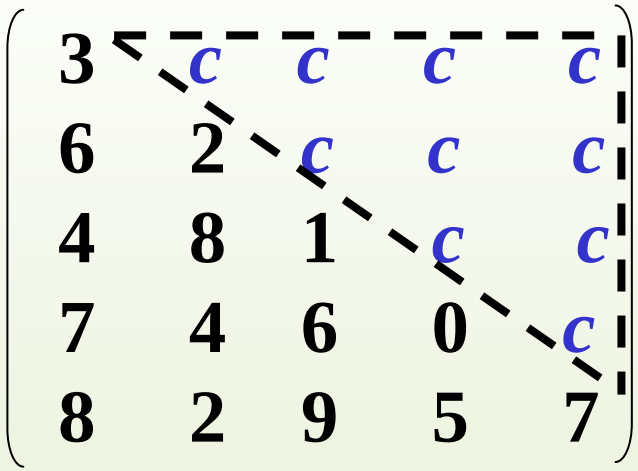


对于下三角中的元素a_{ij}（$i \geqslant j$），在数组SA中的下标k与i、j的关系为：$k=i\times(i+1)/2+j$ 。

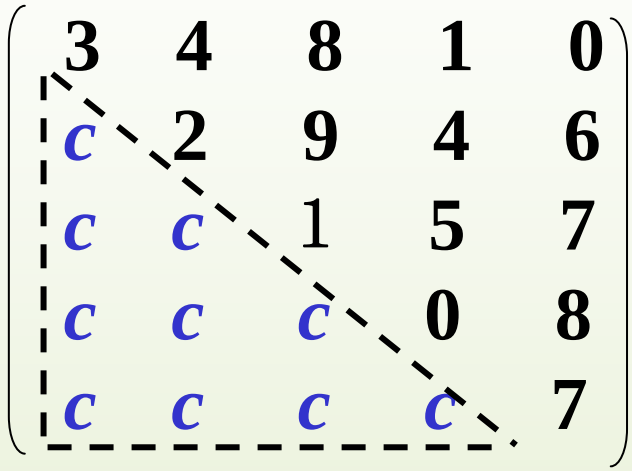
上三角中的元素a_{ij}（$i<j$），因为$a_{ij}=a_{ji}$，则访问和它对应的元素a_{ji}即可，即：$k=j\times(j+1)/2+i$ 。

矩阵的压缩存储

特殊矩阵的压缩存储——三角矩阵



(a) 下三角矩阵



(b) 上三角矩阵

如何压缩存储？

只存储上三角（或下三角）部分的元素。

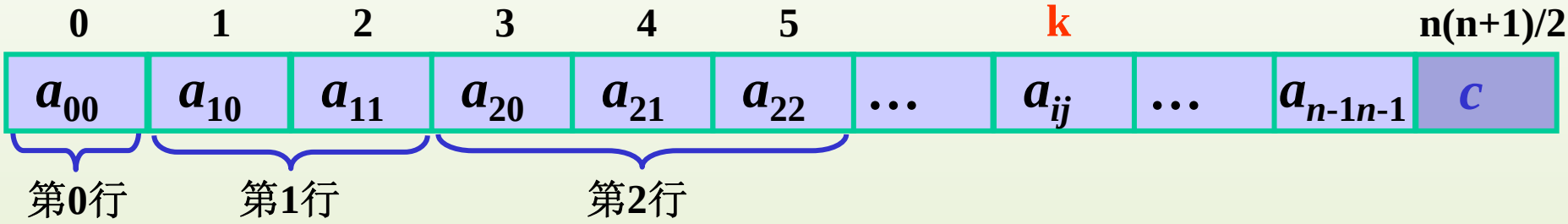
矩阵的压缩存储

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$$

下三角矩阵的压缩存储

存储 { 下三角元素

对角线上方的常数——只存一个



矩阵中任一元素a_{ij}在数组中的下标k与i、j的对应关系：

$$k=\begin{cases} i\times(i+1)/2+j & \text{当} i\geqslant j \\ n\times(n+1)/2 & \text{当} i<j \end{cases}$$

（对角线上方的常数，只存一个）

矩阵的压缩存储

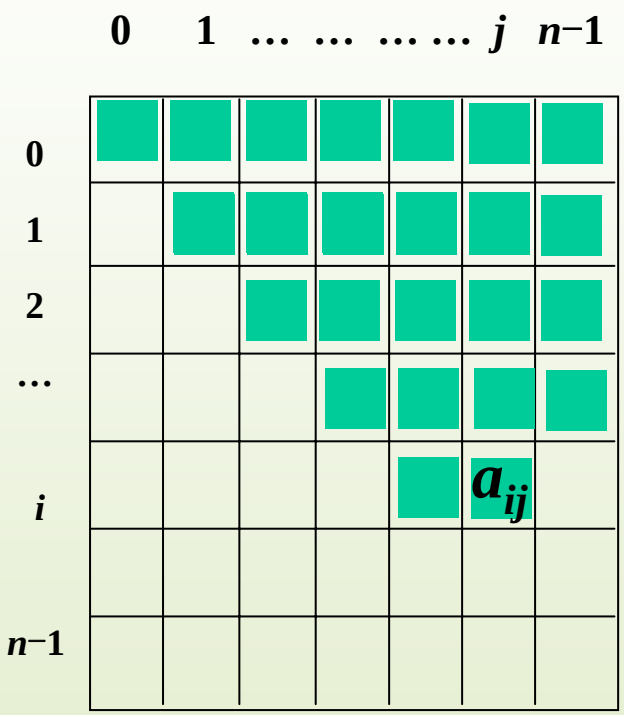
上三角矩阵的压缩存储

存储 {
- 上三角元素
- 对角线下方的常数——只存一个
}

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$$

矩阵的压缩存储

上三角矩阵的压缩存储



每行元素个数

第0行	n
第1行	$n-1$
第2行	$n-2$
…	…
第i-1行	$n-(i-1)$

a_{ij}在本行中的序号$j-i+1$

a_{ij}在一维数组中的序号

$=(n+n-1+n-2+\ldots+n-(i-1))+j-i+1$

$= i\times(2n-i+1)/2+ j-i+1$

∵一维数组下标从0开始

∴a_{ij}在一维数组中的下标

$k= i\times(2n-i+1)/2+ j-i$

(a)上三角矩阵 **(b) 存储说明** **(c) 计算方法**

矩阵的压缩存储

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$$

上三角矩阵的压缩存储

存储 { 上三角元素
对角线下方的常数——只存一个

矩阵中任一元素a_{ij}在数组中的下标k与i、j的对应关系：

$$k=\begin{cases} i\times(2n-i+1)/2+j-i & \text{当} i\leqslant j \\ n\times(n+1)/2 & \text{当} i>j \end{cases}$$

矩阵的压缩存储

特殊矩阵的压缩存储——对角矩阵

对角矩阵：所有非零元素都集中在以主对角线为中心的带状区域中，除了主对角线和它的上下方若干条对角线的元素外，所有其他元素都为零。

$$A=\begin{pmatrix} a_{00} & a_{01} & 0 & 0 & 0 \\ a_{10} & a_{11} & a_{12} & 0 & 0 \\ 0 & a_{21} & a_{22} & a_{23} & 0 \\ 0 & 0 & a_{32} & a_{33} & a_{34} \\ 0 & 0 & 0 & a_{43} & a_{44} \end{pmatrix}$$

矩阵的压缩存储

对角矩阵的压缩存储——压缩到二维数组中

$$A=\begin{pmatrix} a_{00} & a_{01} & 0 & 0 & 0 \\ a_{10} & a_{11} & a_{12} & 0 & 0 \\ 0 & a_{21} & a_{22} & a_{23} & 0 \\ 0 & 0 & a_{32} & a_{33} & a_{34} \\ 0 & 0 & 0 & a_{43} & a_{44} \end{pmatrix}$$

将带状区域立起来

$$B=\begin{pmatrix} 0 & a_{00} & a_{01} \\ a_{10} & a_{11} & a_{12} \\ a_{21} & a_{22} & a_{23} \\ a_{32} & a_{33} & a_{34} \\ a_{43} & a_{44} & 0 \end{pmatrix}$$

映射到二维数组B中，

a_{ij}映射到b_{ts}，映射关系：$\begin{cases} t=i \\ s=j-i+1 \end{cases}$

矩阵的压缩存储

对角矩阵的压缩存储——压缩到一维数组中

$$A=\begin{pmatrix} a_{00} & a_{01} & 0 & 0 & 0 \\ a_{10} & a_{11} & a_{12} & 0 & 0 \\ 0 & a_{21} & a_{22} & a_{23} & 0 \\ 0 & 0 & a_{32} & a_{33} & a_{34} \\ 0 & 0 & 0 & a_{43} & a_{44} \end{pmatrix}$$

(a) 三对角矩阵

元素a_{ij}在一维数组中的序号

$=2+3(i-1)+(j-i+2)$

$=2i+j+1$

∵一维数组下标从0开始

∴元素a_{ij}在一维数组中的下标

$=2i+j$

(b) 寻址的计算方法

按行存储

0	1	2	3	4	5	6	7	8	9	10	11	12
a_{00}	a_{01}	a_{10}	a_{11}	a_{12}	a_{21}	a_{22}	a_{23}	a_{32}	a_{33}	a_{34}	a_{43}	a_{44}

(c) 压缩到一维数组中

矩阵的压缩存储

稀疏矩阵的压缩存储

稀疏矩阵：零元素居多的矩阵，非零元素个数远远小于矩阵元素总数，如：

$$\frac{\text{非零元素个数}}{\text{矩阵元素总数}} < 0.05$$

$$A=\begin{pmatrix} 15 & 0 & 0 & 0 & 0 & 0 \\ 0 & 11 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 9 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

如何只存储非零元素？

注意：稀疏矩阵中的非零元素的分布没有规律。

矩阵的压缩存储

稀疏矩阵的压缩存储

将稀疏矩阵中的每个非零元素表示为：

（行号，列号，非零元素值）——三元组

定义三元组：

```
template <class T>
struct element
{
    int row, col;     //行号，列号
    T item;           //非零元素值
};
```

矩阵的压缩存储

稀疏矩阵的压缩存储

三元组表：将稀疏矩阵的非零元素对应的三元组所构成的集合，按行优先的顺序排列成一个线性表。

$$A=\begin{pmatrix} 15 & 0 & 0 & 0 & 0 & 0 \\ 0 & 11 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 9 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

三元组表＝((0,0,15), (1,1,11), (2,3,6), (4,0,9))

如何存储三元组表？

矩阵的压缩存储

稀疏矩阵的压缩存储——三元组顺序表

采用顺序存储结构存储三元组表

$$A=\begin{pmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

三元组顺序表是否需要预留存储空间？

稀疏矩阵的修改操作

三元组顺序表的插入/删除操作

矩阵的压缩存储

稀疏矩阵的压缩存储——三元组顺序表

采用顺序存储结构存储三元组表

$$A=\begin{pmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

	row	col	item
0	0	0	15
1	0	3	22
2	0	5	−15
3	1	1	11
4	1	2	3
5	2	3	6
6	4	0	91
	空	空	空
MaxTerm−1	闲	闲	闲

5（矩阵的行数）
6（矩阵的列数）
7（非零元个数）

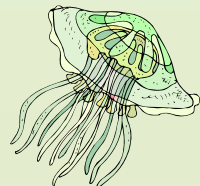
是否对应惟一的稀疏矩阵？

矩阵的压缩存储

稀疏矩阵的压缩存储——三元组顺序表

存储结构定义：

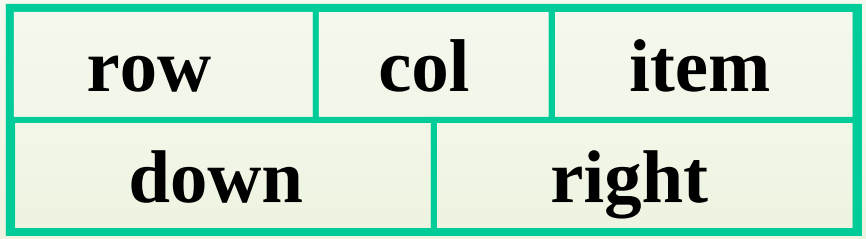
```
const int MaxTerm=100;
template <class T>
struct SparseMatrix
{
   element data[MaxTerm];   //存储非零元素
   int mu, nu, tu;          //行数，列数，非零元个数
};
```



矩阵的压缩存储

稀疏矩阵的压缩存储——十字链表

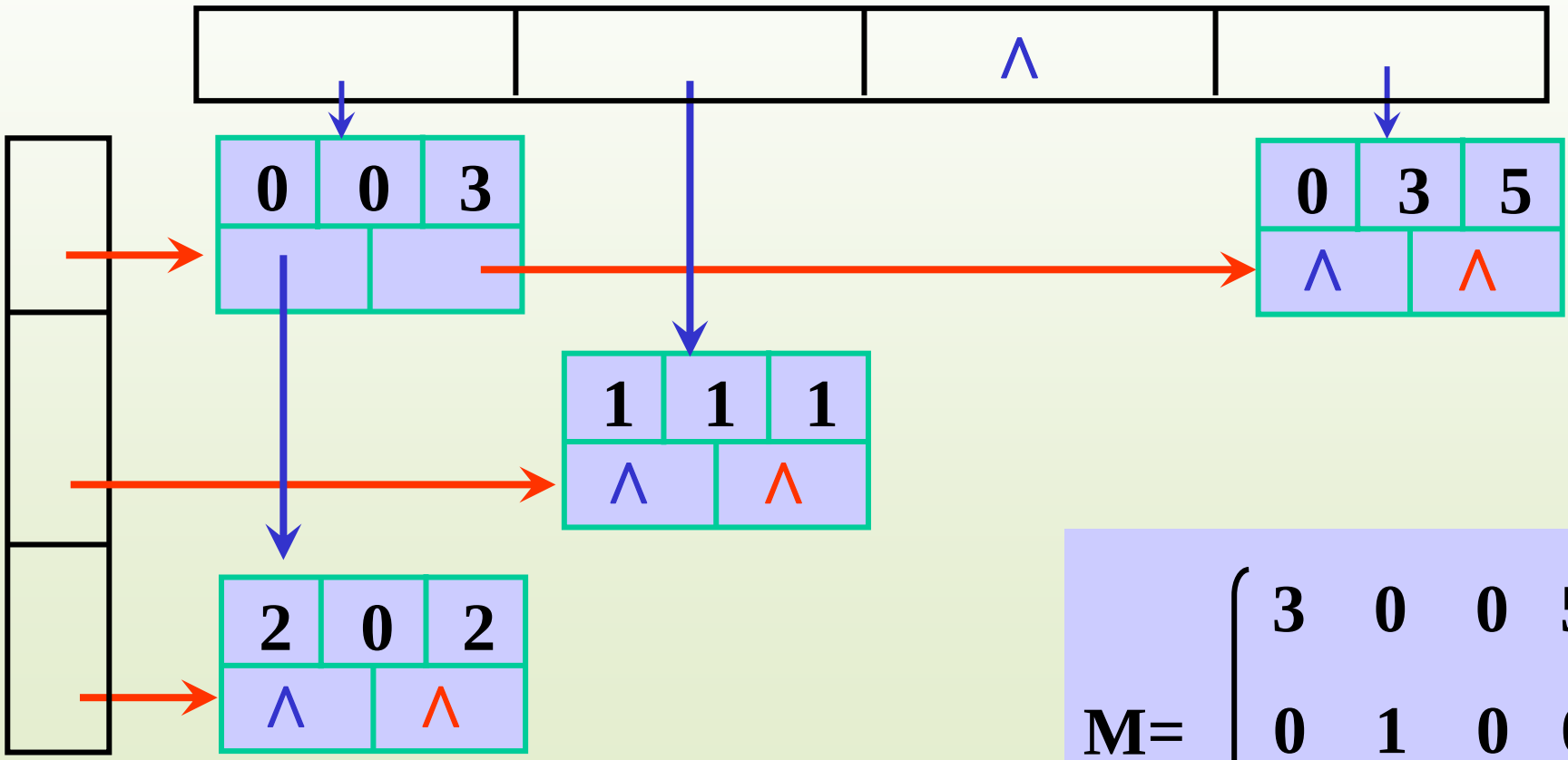
采用链接存储结构存储三元组表，每个非零元素对应的三元组存储为一个链表结点，结构为：



row：存储非零元素的行号
col：存储非零元素的列号
item：存储非零元素的值
right：指针域，指向同一行中的下一个三元组
down：指针域，指向同一列中的下一个三元组

矩阵的压缩存储

稀疏矩阵的压缩存储——十字链表



$$M=\begin{pmatrix} 3 & 0 & 0 & 5 \\ 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 \end{pmatrix}$$

第四章 字符串和多维数组