# 5 数组(矩阵)

董洪伟

http://hwdong.com

# 主要内容

- 数组的类型定义
- 数组的顺序表示和实现
- 矩阵的压缩存储
  - 特殊矩阵
  - 稀疏矩阵

# 数组的类型定义

- 二维数组可以看成数据元素是一维数组的一维数组，即线性表

$$\begin{pmatrix} a_{11} & a_{12} \ldots & a_{1n} \\ a_{11} & a_{12} \ldots & a_{1n} \\ \ldots & & \\ a_{m1} & a_{m2} \ldots & a_{mn} \end{pmatrix} \qquad \begin{aligned} \alpha_1 &= (a_{11} \quad a_{12} \ldots \quad a_{1n}) \\ \alpha_2 &= (a_{21} \quad a_{22} \ldots \quad a_{2n}) \\ &\ldots \\ \alpha_m &= (a_{m1} \quad a_{m2} \ldots \quad a_{mn}) \end{aligned} \qquad A = \begin{pmatrix} \alpha_1 \\ \alpha_2 \\ . \\ . \\ . \\ \alpha_m \end{pmatrix}$$

$$\beta_1 = \begin{pmatrix} a_{11} \\ a_{21} \\ . \\ . \\ . \\ a_{m1} \end{pmatrix} \quad \beta_2 = \begin{pmatrix} a_{12} \\ a_{22} \\ . \\ . \\ . \\ a_{m2} \end{pmatrix} \quad \ldots \quad \beta_n = \begin{pmatrix} a_{1n} \\ a_{2n} \\ . \\ . \\ . \\ a_{mn} \end{pmatrix}$$

$$A = (\beta_1 \quad \beta_2 \ldots \quad \beta_n)$$

# 数组的类型定义

- 数组 是 (一组下标，值) 的集合。

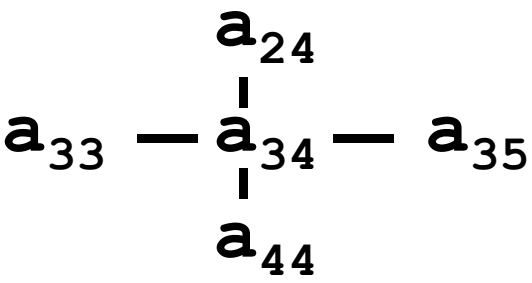
  $A_{7\times 8} = \{a_{ij} \mid i\in[0,6],\ j\in[0,7],\ a_{ij} \in R\}$

- 根据下标读、写相应元素。

  $a_{34} = 10;$

  $b = a_{34}$

- 多维数组的逻辑结构不是非线性而是图型结构



# 数组的类型定义

ADT Array {

D＝{$a_{j_1 j_2 \cdots j_n}$ | $j_i$ =0, 1, . . . , $b_i$-1,
i=1, 2, . . , n, n(>0)称为数组的维数, $b_i$是数组第i维的长度, $j_i$是数组元素的第i维下标}

基本操作:

InitArray(&A, n, $b_1$, . . . , $b_n$);
//由维数n和各维长度构造相应的数组A

DestroyArray(&A); //销毁数组A。

Value(A, &e, $i_1$, . . . , $i_n$)
//查询数组A下标为$i_1$, . . . , $i_n$的数组元素的值

Assign(&A, e, $i_1$, . . . , $i_n$)
//对数组A下标为$i_1$, . . . , $i_n$的数组元素进行赋值

**} ADT Array**

# C语言的二维数组

## Initializing Two-Dimensional Arrays:

```
int a[3][4] = { {0, I, 2, 3} , /* initializers for row indexed by 0 */
                {4, 5, 6, 7} , /* initializers for row indexed by I */
                {8, 9, I0, II}};  /* initializers for row indexed by 2 */
```

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

## Accessing Two-Dimensional Array Elements:

```
int val = a[2][3];
a[1][2] = 34;
```

# 数组的顺序表示和实现

- 类型特点：

  1） 没有插入、删除操作，适合顺序存储；

  2） 数组是多维的结构，而存储空间是一个一维的结构。

- 两种顺序映象的方式：

  1) 以行序为主序(低下标优先)；

  $A_{00}$ $A_{01}$ $A_{10}$ $A_{11}$

  2） 以列序为主序(高下标优先)；

  $A_{00}$ $A_{10}$ $A_{01}$ $A_{11}$

# 一维数组

- 一维数组在内存中的存放
  - 假设**NumberA**的首地址为**12**
    - 第**i**个元素的地址 = 首地址 + **i***数据类型的大小
    - 假设一个单元的大小是**4**个字节

| 下标： | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 值： | 5 | 3 | 11 | 2 |

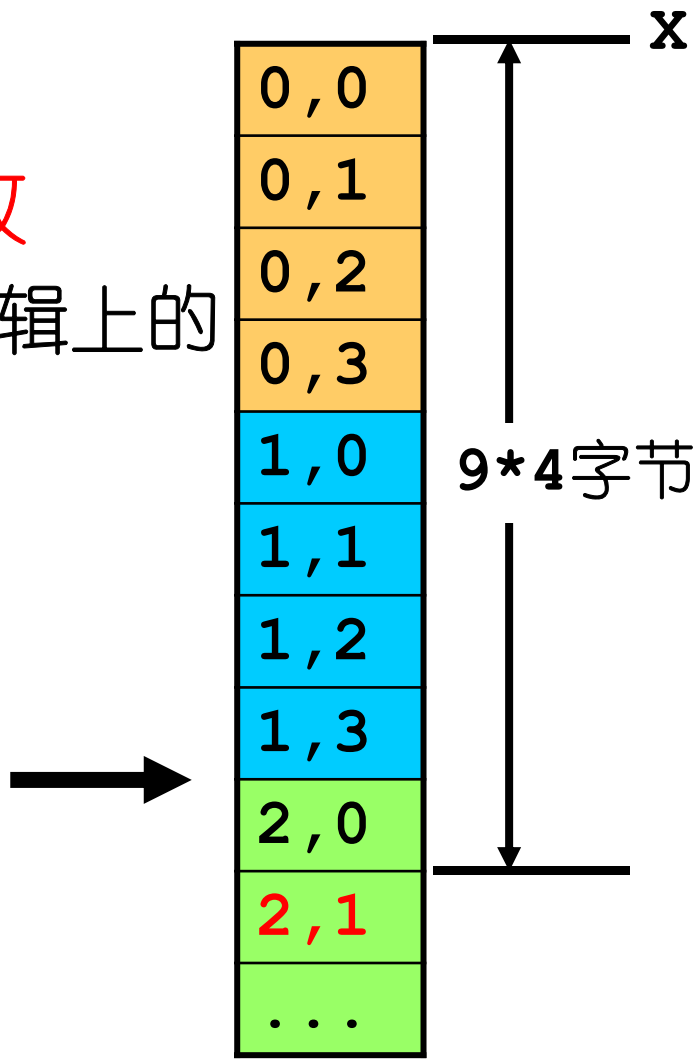
| 地址： | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

# 二维数组

- 二维数组在内存中的存放
  - 二维数组的“二维”是逻辑上的
  - 内存永远是线性编址

| 0,0 | 0,1 | 0,2 | 0,3 |
|---|---|---|---|
| 1,0 | 1,1 | 1,2 | 1,3 |
| 2,0 | 2,1 | 2,2 | 2,3 |
| 3,0 | 3,1 | 3,2 | 3,3 |
| 4,0 | 4,1 | 4,2 | 4,3 |

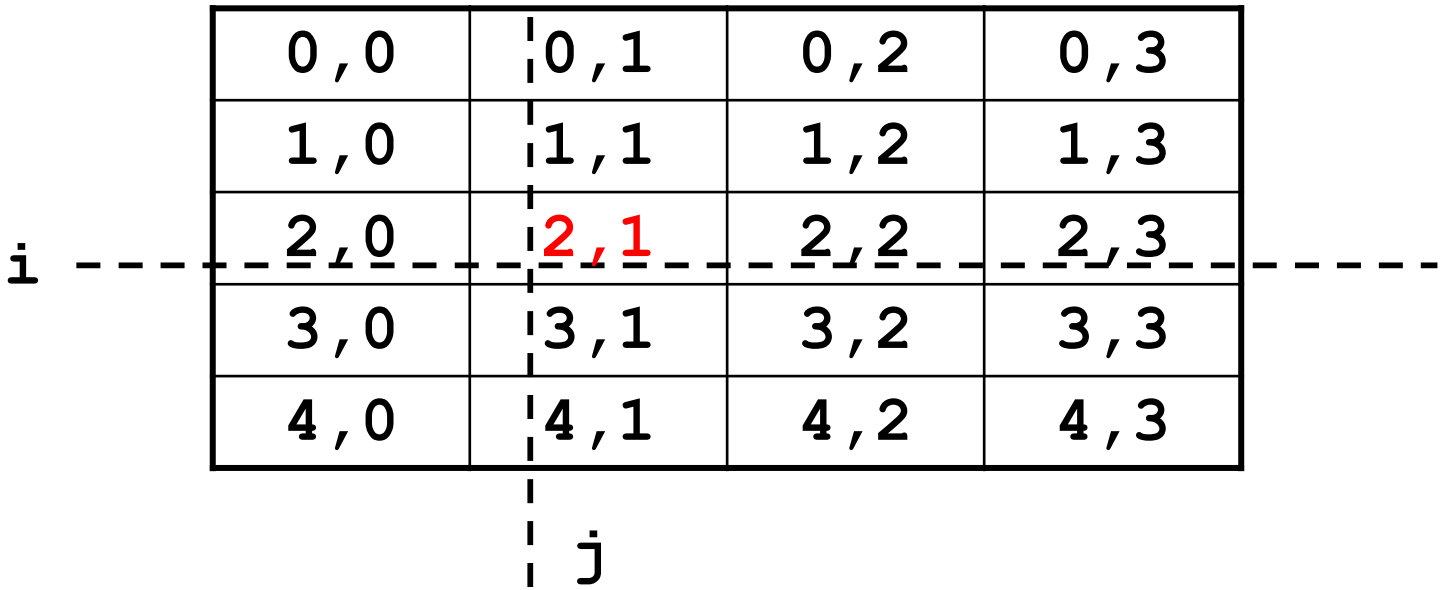
“二维数组”




内存中的存放方式

# 二维数组

- 按照行优先，Data[i][j]是第几个元素？
  - Data[i][j]前面有i行，j列
  - 所以Data[i][j]是第“i*列数+j”个元素(从0开始)
  - 所以Data[i][j]的地址=首地址+(i*列数+j)*数据类型的大小



# 二维数组

- 二维数组**A**中任一元素$a_{ij}$在内存中的位置

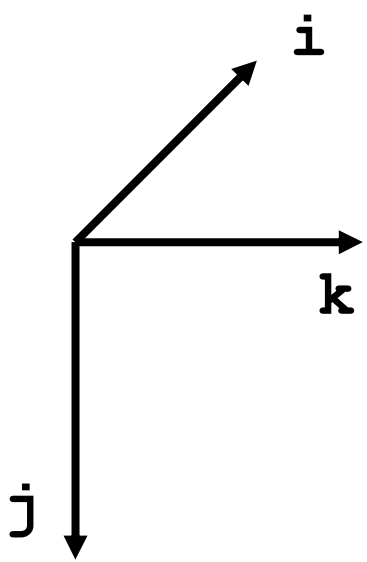
$$LOC(i, j) = LOC(0, 0) + (b_2 \times i + j)L$$

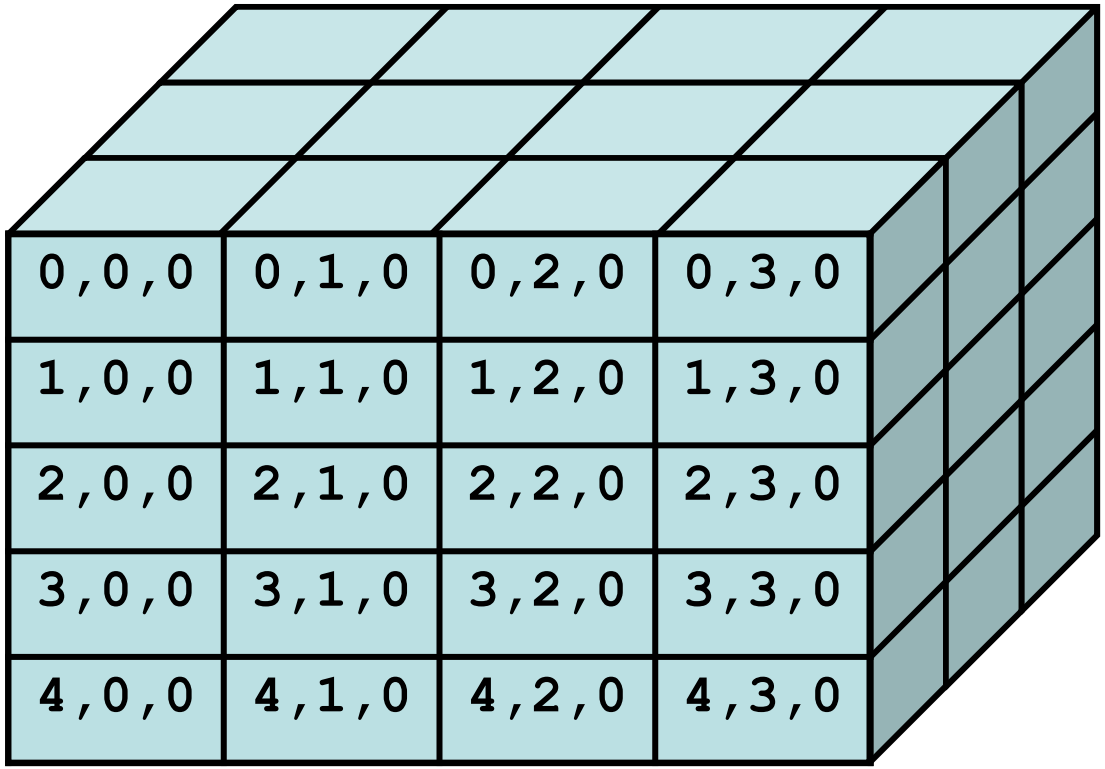
- **N**维数组**A**中任一元素在内存中的位置

$$LOC(j_1, j_2, \ldots j_n) = LOC(0, 0, \ldots 0)$$
$$+ (b_2 \times \ldots \times b_n \times j_1 + b_3 \times \ldots \times b_n \times j_2$$
$$+ \ldots + b_n \times j_{n-1} + j_n)L$$
$$= LOC(0, 0, \ldots 0) + \left( \sum_{i=1}^{n-1} j_i \prod_{k=i+1}^{n} b_k + j_n \right) L$$

# N维数组

$A_{3*5*4}$




$a_{i,j,k}$ i*5*4+j*4+k

$a_{2,3,3}$

# 下标--地址映射—n维情形

设 n 维数组$A_{b1\times b2\times\ldots\times bn}$，则：

$$LOC(j_1, j_2, \ldots, j_n)$$

$$= LOC(0, 0, \ldots, 0)$$

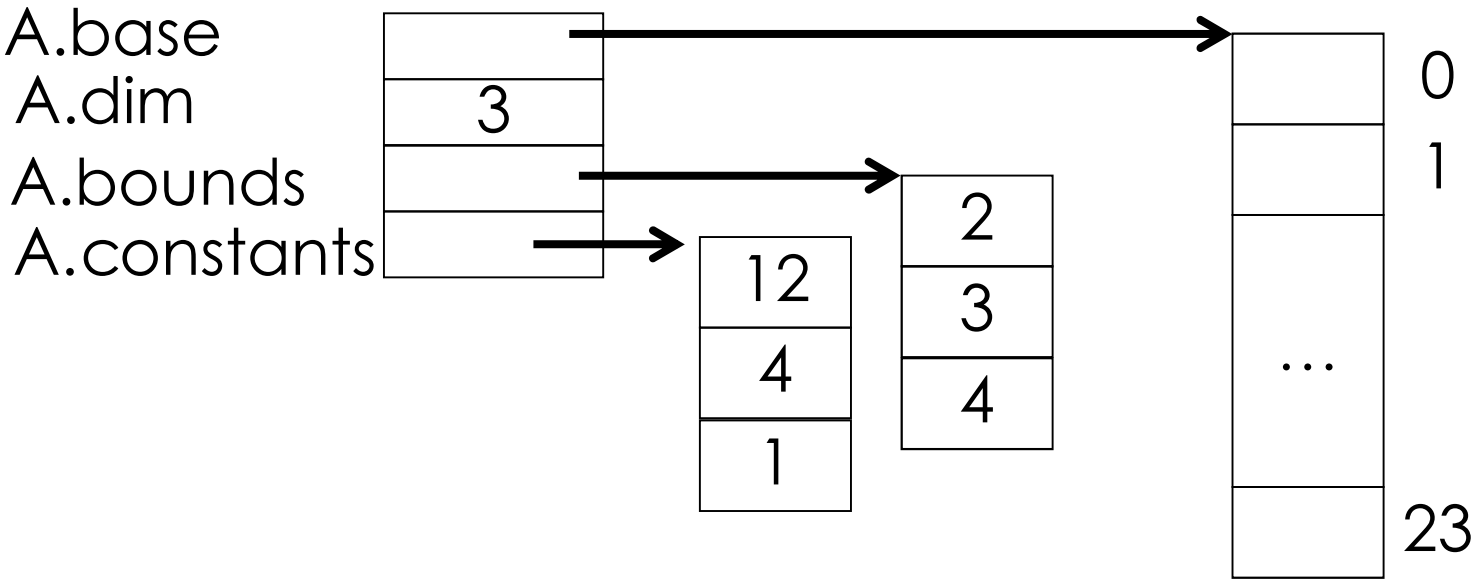
$$+ \underbrace{j_1*b_2*\ldots*b_n*L}_{c_1} + \underbrace{j_2*b_3*\ldots*b_n*L}_{c_2} + \ldots + \underbrace{j_{n-1}*b_n*L}_{c_{n-1}} + \underbrace{j_n*L}_{c_n}$$

$$= LOC(0, 0, \ldots, 0) + \sum_{i=1}^{n}(c_i * j_i)$$

其中，$c_n=L$，$c_{i-1}=c_i*b_i$，$1<i\leqslant n$，称求址常量。

# 数组的表示与实现

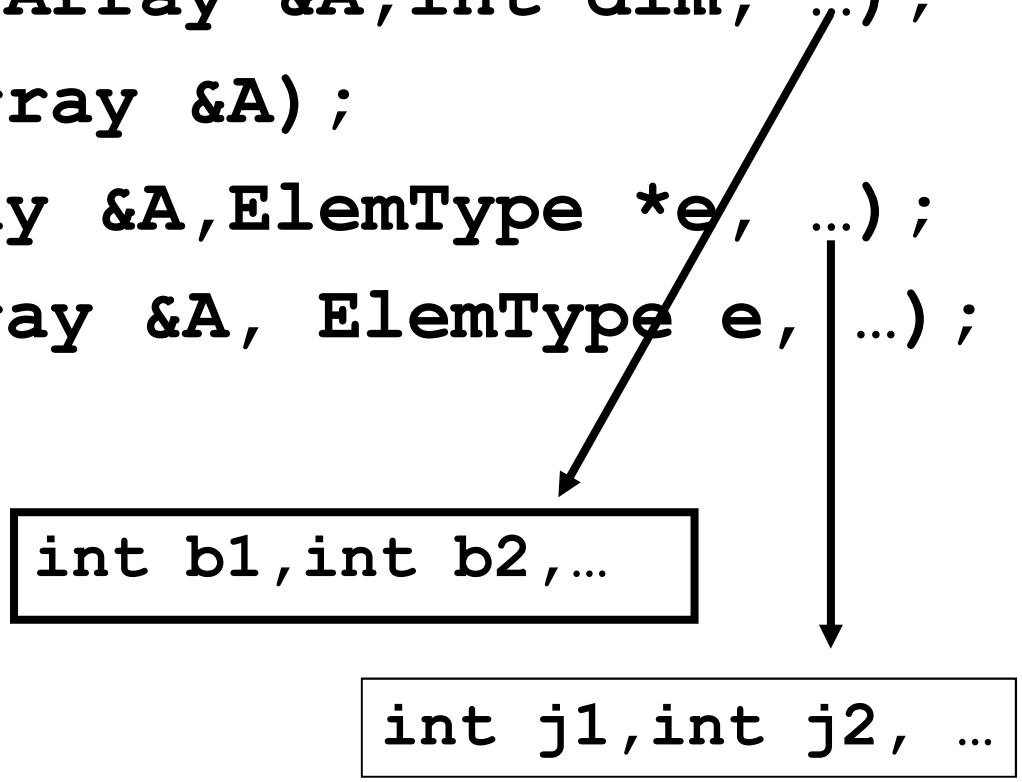
```
typedef struct{
   ElemType    *base;//数组的基地址
   int dim;                  //维数
   int *bounds;    //各维的界
   int *constants;//地址函数的系数,即Ci
}Array;
```



# 数组的表示与实现

基本操作的实现：

```
bool  InitArray(Array &A,int dim, …);
bool  Destory(Array &A);
bool  Value(Array &A,ElemType *e, …);
bool  Assign(Array &A, ElemType e, …);
```



# 数组的表示与实现：InitArray

```
bool InitArray(Array &A,int dim, ...)
{
  if( dim <1||dim >= Max_Array_Dim)
      return false;//维数非法
  A.dim = dim;
  A.bounds = (int *)malloc(dim*
                  sizeof(int));
  if( !A.bounds )
       return ERROR;//内存分配失败
```

# 数组的表示与实现：InitArray

```
 int elemtotal = 1;  //元素的总数
va_list ap; va_start(ap,dim);
for( int i=0; i<dim; ++i){
   A.bounds[i] = va_arg ( ap,int);
   if(A.bounds[i] <=0) return -1;
   elemtotal *= A.bounds[i] ;
}
va_end( ap );

 A.base = (ElemType *)malloc(
        elemtotal *sizeof(ElemType));
if(!A.base) return ERROR;
```

# 数组的表示与实现：InitArray

```
  A.constants=(int *)malloc( dim *
                  sizeof(int)); //计算Ci
  if (!A.constants)
       return ERROR;     //存储分配失败
  A.constants[dim-1]=1;          //cn=1
  for(i= dim-2;i>=0;--i) //ci=ci+1*bi+1
      A.constants[i] = A.bounds[i+1]
                          *A.constants[i+1];
  return OK;
}
```

# va...实现可变的参数表

```
void va_start (va_list param,lastfix);
```

param指向被调用函数的固定参数lastfix 后的可变参数列表。

```
type va_arg (va_list param,type);
```

从param所指处取出一个类型为type的参数并返回该值。

```
void va_end (va_list param);
```

param列表解析的结束。

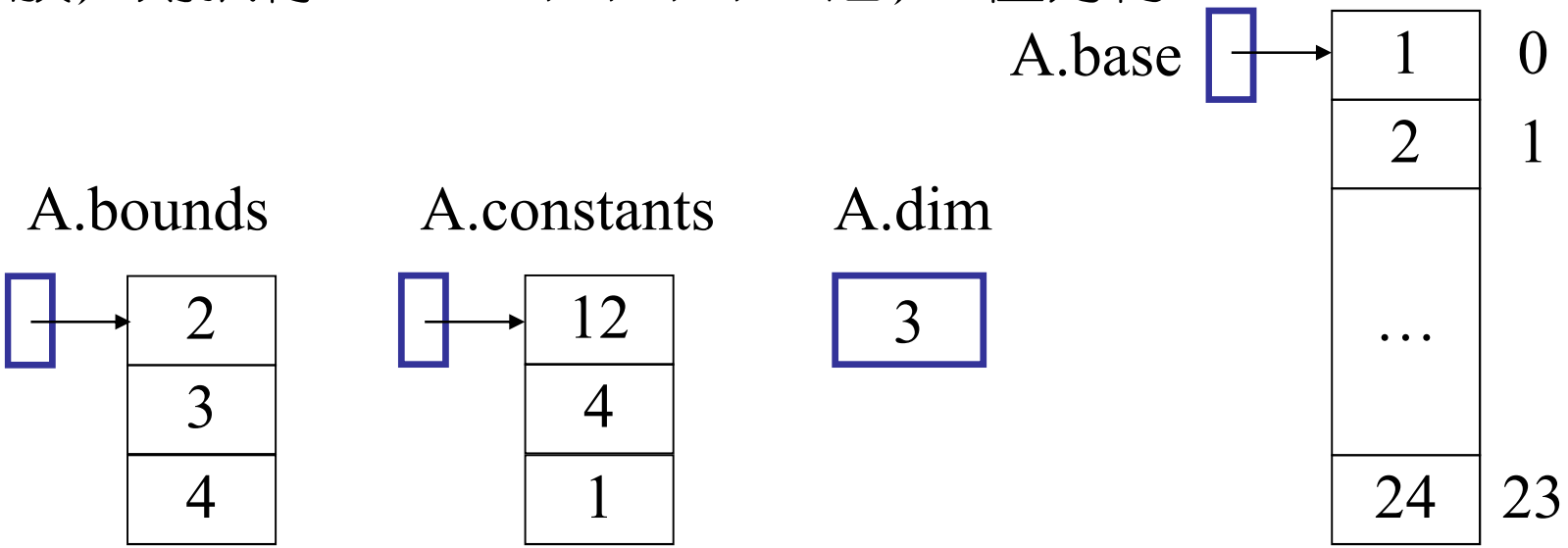
# 数组的表示与实现：取数组元素的值

【思路】

(1)按下标求数据元素在存储空间中的相对位置off $=\sum_{i=1}^{n}(c_i * j_i)$

(2)e=*(A.base+off)。

【例】设数组A=(((1,2,3,4),(5,6,7,8),(9,10,11,12)),((13,14,15,16),(17,18,19,20),(21,22,23,24)))按行序存放，则执行Value(A,e,0,2,1)后，e值为何？



# 数组的表示与实现:取数组元素的值

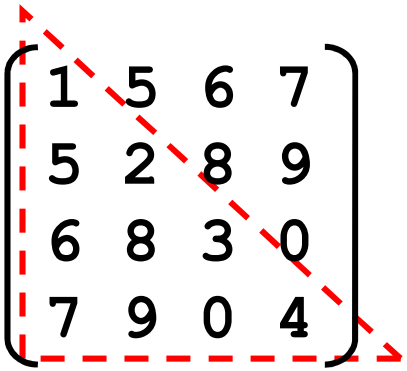
```
bool Value(Array &A,ElemType *e,...){
  int off=0; va_list ap;
  va_start(ap,e);    //使ap指向e后第一个参数
  for(int i=0;i<A.dim;i++){
   int j=va_arg(ap,int);    //j中保存当前下标
   if(j<0||j>=A.bounds[i])
       return ERROR;              //下标值非法
   off+=A.constants[i]*j;    //求ci*ji并累加
  }//for
  va_end(ap);
```

# 数组的表示与实现:取数组元素的值

```
    *e=*(A.base+off); //用e返回元素值
    return OK;
}//Value
```

# 矩阵的压缩

- 特殊矩阵
  - 对称矩阵
    - 即$a_{ij}=a_{ji}$，1<=i，j<=n
    - 可以将$n^2$个单元压缩为n(n+1)/2个：

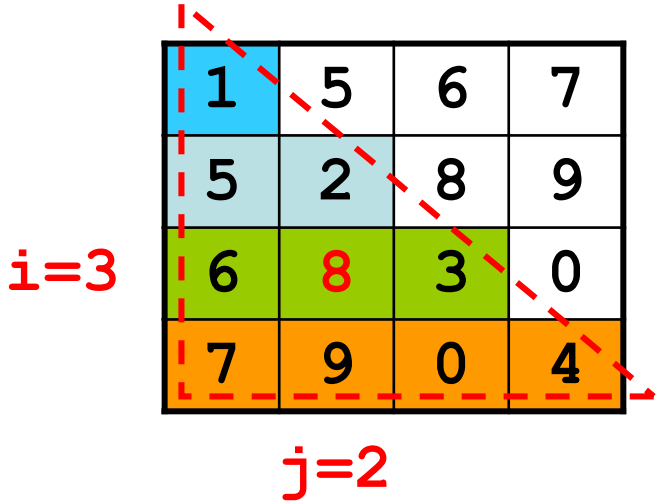



| 1 | 5 | 6 | 7 |
|---|---|---|---|
| 5 | 2 | 8 | 9 |
| 6 | 8 | 3 | 0 |
| 7 | 9 | 0 | 4 |

| 1 | 5 | 2 | 6 | 8 | 3 | 7 | 9 | 0 | 4 |
|---|---|---|---|---|---|---|---|---|---|

- k的含义：按行优先，是第k个（从0开始）

$$k=\begin{cases}\dfrac{i(i-1)}{2}+j-1 & i\geq j \quad \text{(保存下三角)}\\ \dfrac{j(j-1)}{2}+i-1 & i<j \quad \text{(保存上三角)}\end{cases}$$

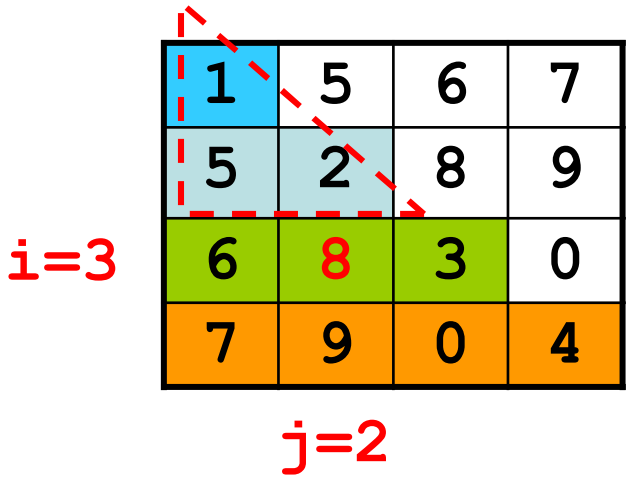



k=4

| 1 | 5 | 2 | 6 | 8 | 3 | 7 | 9 | 0 | 4 |
|---|---|---|---|---|---|---|---|---|---|

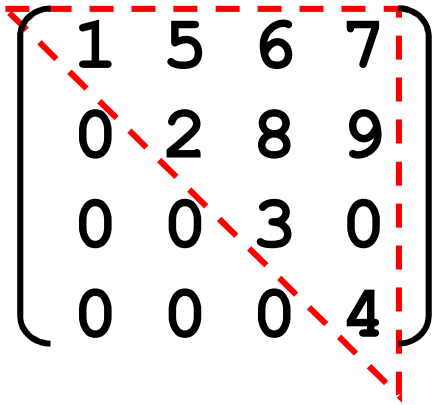
# 矩阵的压缩

- 公式的推导（下三角）
  - **i=3，j=2**
  - 则前面有一个**i-1**行的完整三角形，共有元素**(1+i-1)(i-1)/2 = i(i-1)/2**个
  - 另外，同一行，前面还有**j-1**个元素
  - 所以，**k = i(i-1)/2+j-1**

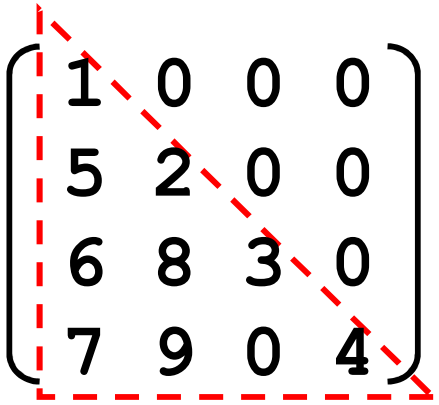


# 矩阵的压缩

- 三角矩阵
  - 压缩方法和对称矩阵完全相同




上三角




下三角

# 矩阵的压缩

- 稀疏矩阵

$$A_{9\times 7} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 7 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

– 非零元素个数远远少于矩阵元素总数

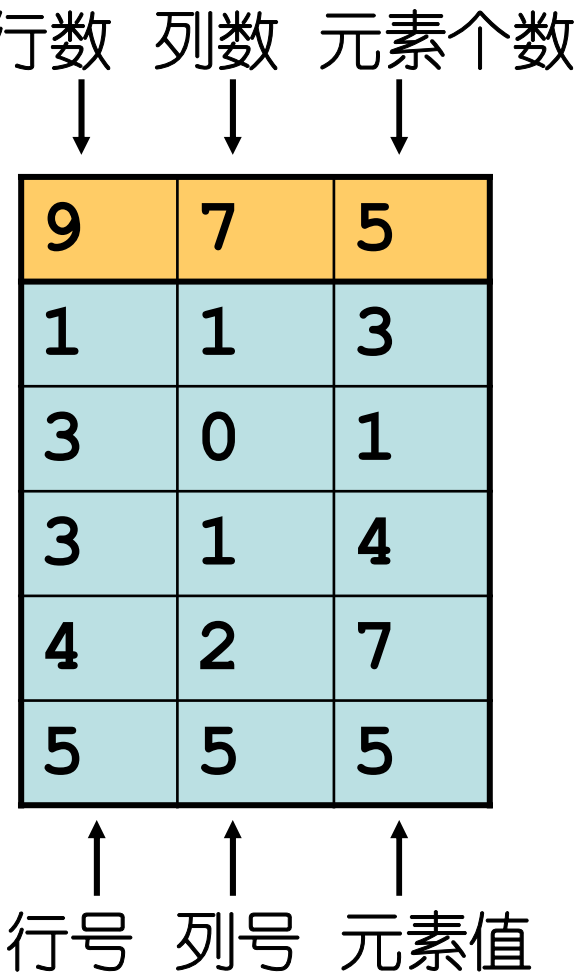
# 矩阵的压缩

- 稀疏矩阵的定义
  - 设矩阵 $A_{m\times n}$ 中有 t 个非零元素，若 t 远远小于矩阵元素的总数 m×n，且非零元素的分布无规律，则称矩阵 A 为稀疏矩阵
  - 为节省存储空间，应只存储非零元素
  - 非零元素的分布一般没有规律，应在存储非零元素时，同时存储该非零元素的行下标 row、列下标 col、值 value
  - 每一个非零元素由一个三元组唯一确定：
    - ( 行号 row, 列号 col, 值 value )

# 矩阵的压缩

- 稀疏矩阵的压缩表示

$$A_{9\times7}=\begin{pmatrix}0&0&0&0&0&0&0\\0&3&0&0&0&0&0\\0&0&0&0&0&0&0\\1&4&0&0&0&0&0\\0&0&7&0&0&0&0\\0&0&0&0&0&5&0\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\end{pmatrix}$$



# 稀疏矩阵的转置

- 三元组顺序表

```
#define MAXSIZE 12500
typedef struct{
    int  i,j;          //行下标、列下标
    ElemType e;        //元素的值
}Triple;

typedef struct{
    Triple data[MAXSIZE+1];
    int mu,nu,tu;      //行数、列数、非0元素个数
}TSMatrix;
```

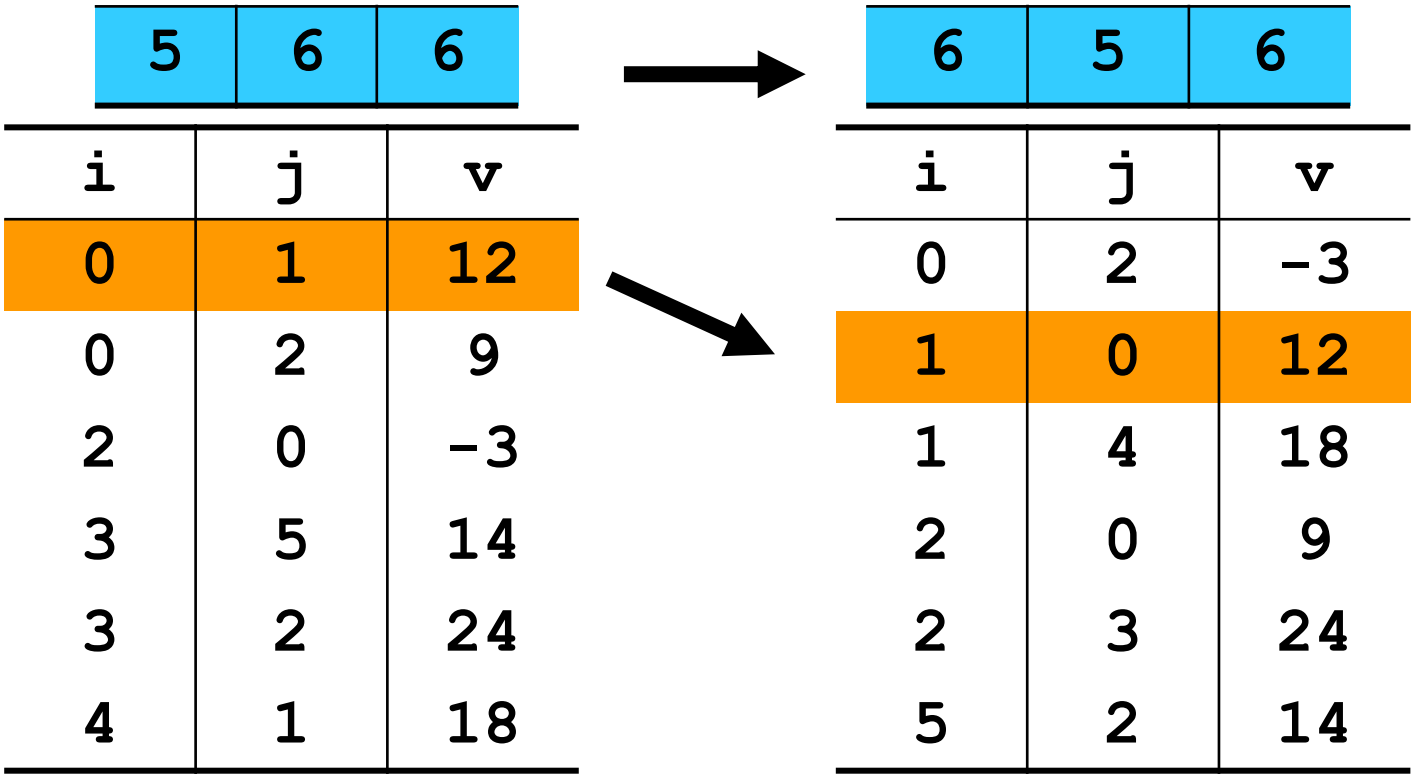
# 稀疏矩阵的转置

- 矩阵的转置
  - $T_{ij} = M_{ij}$

| | | | | | |
|---|---|---|---|---|---|
| 0 | 12 | 9 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 |
| -3 | 0 | 0 | 0 | 0 | 14 |
| 0 | 0 | 24 | 0 | 0 | 0 |
| 0 | 18 | 0 | 0 | 0 | 0 |

| | | | | |
|---|---|---|---|---|
| 0 | 0 | -3 | 0 | 0 |
| 12 | 0 | 0 | 0 | 18 |
| 9 | 0 | 0 | 24 | 0 |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 18 | 0 | 0 | 0 |
| 0 | 0 | 14 | 0 | 0 |

- 行数和列数交换
- i、j的值相互交换
- 重排三元组之间的次序



| 5 | 6 | 6 |
|---|---|---|
| i | j | v |
| 0 | 1 | 12 |
| 0 | 2 | 9 |
| 2 | 0 | -3 |
| 3 | 5 | 14 |
| 3 | 2 | 24 |
| 4 | 1 | 18 |

| 6 | 5 | 6 |
|---|---|---|
| i | j | v |
| 0 | 2 | -3 |
| 1 | 0 | 12 |
| 1 | 4 | 18 |
| 2 | 0 | 9 |
| 2 | 3 | 24 |
| 5 | 2 | 14 |

- 简单算法

```
void transpose(TSMatrix &T,const TSMatrix &M)
{
   T.mu = M.nu; T.nu = M.mn; T.tu = M.tu;
   if(T.tu) {
      q = 0;
      for(col = 0; col < M.nu; col ++)
         for(p = 0; p < M.tu; p ++)
            if(M.data[p].j == col) {
               T.data[q].i = M.data[p].j;
               T.data[q].j = M.data[p].i;
               T.data[q].e = M.data[p].e;
               q ++;
            }
    }
}
```

复制总体信息

```
for(col = 0; col < M.nu; col ++)
    for(p = 0; p < M.tu; p ++)
        if(M.data[p].j == col) {
            T.data[q].i = M.data[p].j;
            T.data[q].j = M.data[p].i;
            T.data[q].e = M.data[p].e;
            q ++;
        }
```

p扫描三元组M的

如果p指向的记录的列下标=col

把M中的记录p复制到T中的记录q

M

| 5 | 6 | 8 |
|---|---|---|

T

| 6 | 5 | 8 |
|---|---|---|

| | i | j | v |
|---|---|---|---|
| p → | 0 | 1 | 12 |
| | 0 | 2 | 9 |
| | 2 | 0 | -3 |
| | 2 | 5 | 14 |

| | i | j | v |
|---|---|---|---|
| q → | | | |
| | | | |
| | | | |
| | | | |

col = 1

```
for(col = 0; col < M.nu; col ++)
    for(p = 0; p < M.tu; p ++)
        if(M.data[p].j == col) {
            T.data[q].i = M.data[p].j;
            T.data[q].j = M.data[p].i;
            T.data[q].e = M.data[p].e;
            q ++;
        }
```

M

| 5 | 6 | 8 |
|---|---|---|

T

| 6 | 5 | 8 |
|---|---|---|

| | i | j | v |
|---|---|---|---|
| | 0 | 1 | 12 |
| | 0 | 2 | 9 |
| p → | 2 | 0 | -3 |
| | 2 | 5 | 14 |

q →

| i | j | v |
|---|---|---|
| 0 | 2 | -3 |
| | | |
| | | |
| | | |

col = 1

```
for(col = 0; col < M.nu; col ++)
    for(p = 0; p < M.tu; p ++)
        if(M.data[p].j == col) {
            T.data[q].i = M.data[p].j;
            T.data[q].j = M.data[p].i;
            T.data[q].e = M.data[p].e;
            q ++;
        }
```

col扫描每一列

M

| 5 | 6 | 8 |
|---|---|---|

T

| 6 | 5 | 8 |
|---|---|---|

| | i | j | v |
|---|---|---|---|
| p → | 0 | 1 | 12 |
| | 0 | 2 | 9 |
| | 2 | 0 | -3 |
| | 2 | 5 | 14 |

| | i | j | v |
|---|---|---|---|
| | 0 | 2 | -3 |
| q → | | | |
| | | | |
| | | | |

col = 2

```
for(col = 0; col < M.nu; col ++)
    for(p = 0; p < M.tu; p ++)
        if(M.data[p].j == col) {
            T.data[q].i = M.data[p].j;
            T.data[q].j = M.data[p].i;
            T.data[q].e = M.data[p].e;
            q ++;
        }
```

col扫描每一列

M

| 5 | 6 | 8 |
|---|---|---|

T

| 6 | 5 | 8 |
|---|---|---|

| | i | j | v |
|---|---|---|---|
| p → | 0 | 1 | 12 |
| | 0 | 2 | 9 |
| | 2 | 0 | -3 |
| | 2 | 5 | 14 |

| | i | j | v |
|---|---|---|---|
| | 0 | 2 | -3 |
| | 1 | 0 | 12 |
| q → | | | |
| | | | |

col = 2

# 稀疏矩阵的转置：简单算法

- 简单算法的分析
  - 稀疏矩阵转置算法复杂度 = O(nu*tu)
  - 比较一般矩阵的转置算法：

```
for(col = 1; col <= nu; col ++)
    for(row = 1; row <= mu; row ++)
        T[col][row] = M[row][col];
```

    - 其复杂度 = O(mu*nu)
  - 如果tu和mu*nu一个数量级，则稀疏矩阵转置算法复杂度=O($mu*nu^2$)
  - 所以此算法只适用于tu<<mu*nu时

# 稀疏矩阵的转置：简单算法

- 简单算法的分析
  - 简单算法的主要问题在于为了使得结果三元组表的记录按照行顺序排列，需要分别针对结果三元组表的每一行（也就是原三元组表的每一列），扫描整个三元组表，查找属于该行的所有记录
  - 即对原三元组表的扫描需要反复进行，而不是只需一次
  - 因此时间复杂度=O(nu*tu)

# 稀疏矩阵的转置：快速转置算法

- 关键：希望对原三元组表只需扫描一遍
- 问题：原三元组表中的一个三元组应该放在结果三元组表中的什么位置呢？

| | 5 | 6 | 6 |
|---|---|---|---|
| | i | j | v |
| 1 | 0 | 1 | 12 |
| 2 | 0 | 2 | 9 |
| 3 | 2 | 0 | -3 |
| 4 | 2 | 5 | 14 |
| 5 | 3 | 2 | 24 |
| 6 | 4 | 1 | 18 |

| | 6 | 5 | 6 |
|---|---|---|---|
| | i | j | v |
| 1 | 0 | 2 | -3 |
| 2 | 1 | 0 | 12 |
| 3 | 1 | 4 | 18 |
| 4 | 2 | 0 | 9 |
| 5 | 2 | 3 | 24 |
| 6 | 5 | 2 | 14 |

# 稀疏矩阵的转置：快速转置算法

- 原每一列在转置后三元组的起始位置
  - 数组**num[col]**：原矩阵第**col**列中，非零元素的个数
  - 数组**cpot[col]**：原矩阵第**col**列中，第**1**个非零元素在结果三元组表中的位置
  - 显然有：

$$\begin{cases} cpot[1] = 1 \\ cpot[col] = cpot[col-1] + num[col-1] \end{cases}$$

# 稀疏矩阵的转置：快速转置算法

| | 5 | 6 | 6 |
|---|---|---|---|
| | i | j | v |
| 0 | 0 | 1 | 12 |
| 1 | 0 | 2 | 9 |
| 2 | 2 | 0 | -3 |
| 3 | 2 | 5 | 14 |
| 4 | 3 | 2 | 24 |
| 5 | 4 | 1 | 18 |

| | 6 | 5 | 6 |
|---|---|---|---|
| | i | j | v |
| 0 | 0 | 2 | -3 |
| 1 | 1 | 0 | 12 |
| 2 | 1 | 4 | 18 |
| 3 | 2 | 0 | 9 |
| 4 | 2 | 3 | 24 |
| 5 | 5 | 2 | 14 |

| col | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| num[col] | 1 | 2 | 2 | 0 | 0 | 1 |
| cpot[col] | 0 | 1 | 3 | 5 | 5 | 6 |

- 快速转置算法

```
void FastTranspose(TSMatrix &T,const TSMatrix
  &M) {
  T.mu = M.nu; T.nu = M.mn; T.tu = M.tu;
  if(T.tu) {
     for(col = 0; col < M.nu; col ++)
        num[col] = 0;              //初始化num
     for(t = 0; t < M.tu; t ++)
        num[M.data[t].j] ++;   //统计每列元素个数
     //计算T中每行开始的位置
     cpot[0] = 1;
     for(col = 1; col < M.nu; col ++)
        cpot[col] = cpot[col-1] + num[col-1];
     ...
```

- 快速转置算法（接上页）

```
        for (p = 0; p < M.tu; p ++) {
            col = M.data[p].j;
            q   = cpot[col];
            T.data[q].i = M.data[p].j;
            T.data[q].j = M.data[p].i;
            T.data[q].e = M.data[p].e;
            cpot[col]++;
        }
    }
}
```

原三元组表只需要扫描一遍

得到当前三元组的列号

得到当前三元组的放置位置

拷贝三元组

当前列的起始位置加1

```
for (p = 0; p < M.tu; p ++) {
   col = M.data[p].j;     q   = cpot[col];
   T.data[q].i = M.data[p].j;
   T.data[q].j = M.data[p].i;
   T.data[q].e = M.data[p].e;
   cpot[col]++; }
```

| | | i | j | v |
|---|---|---|---|---|
| p → | 0 | 0 | 1 | 12 |
| | 1 | 0 | 2 | 9 |
| | 2 | 2 | 0 | -3 |
| | 3 | 2 | 5 | 14 |
| | 4 | 3 | 2 | 24 |
| | 5 | 4 | 1 | 18 |

| | | i | j | v |
|---|---|---|---|---|
| | 0 | | | |
| q → | 1 | 1 | 0 | 12 |
| | 2 | | | |
| | 3 | | | |
| | 4 | | | |
| | 5 | | | |

col ↓

| col | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| num[col] | 1 | 2 | 2 | 0 | 0 | 1 |
| cpot[col] | 0 | 1 | 3 | 5 | 5 | 5 |

```
for (p = 0; p < M.tu; p ++) {
   col = M.data[p].j;     q   = cpot[col];
   T.data[q].i = M.data[p].j;
   T.data[q].j = M.data[p].i;
   T.data[q].e = M.data[p].e;
   cpot[col]++; }
```

| | i | j | v |
|---|---|---|---|
| 0 | 0 | 1 | 12 |
| p → 1 | 0 | 2 | 9 |
| 2 | 2 | 0 | -3 |
| 3 | 2 | 5 | 14 |
| 4 | 3 | 2 | 24 |
| 5 | 4 | 1 | 18 |

| | i | j | v |
|---|---|---|---|
| 0 | | | |
| 1 | 1 | 0 | 12 |
| 2 | | | |
| q → 3 | 2 | 0 | 9 |
| 4 | | | |
| 5 | | | |

col ↓

| col | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| num[col] | 1 | 2 | 2 | 0 | 0 | 1 |
| cpot[col] | 0 | 2 | 3 | 5 | 5 | 5 |

```
for (p = 0; p < M.tu; p ++) {
   col = M.data[p].j;     q   = cpot[col];
   T.data[q].i = M.data[p].j;
   T.data[q].j = M.data[p].i;
   T.data[q].e = M.data[p].e;
   cpot[col]++; }
```

| | i | j | v |
|---|---|---|---|
| 0 | 0 | 1 | 12 |
| p → 1 | 0 | 2 | 9 |
| 2 | 2 | 0 | -3 |
| 3 | 2 | 5 | 14 |
| 4 | 3 | 2 | 24 |
| 5 | 4 | 1 | 18 |

| | i | j | v |
|---|---|---|---|
| 0 | | | |
| 1 | 1 | 0 | 12 |
| 2 | | | |
| q → 3 | 2 | 0 | 9 |
| 4 | | | |
| 5 | | | |

col ↓

| col | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| num[col] | 1 | 2 | 2 | 0 | 0 | 1 |
| cpot[col] | 0 | 2 | 4 | 5 | 5 | 5 |

```
for (p = 0; p < M.tu; p ++) {
    col = M.data[p].j;     q   = cpot[col];
    T.data[q].i = M.data[p].j;
    T.data[q].j = M.data[p].i;
    T.data[q].e = M.data[p].e;
    cpot[col]++; }
```

p →

| | i | j | v |
|---|---|---|---|
| 0 | 0 | 1 | 12 |
| 1 | 0 | 2 | 9 |
| 2 | 2 | 0 | -3 |
| 3 | 2 | 5 | 14 |
| 4 | 3 | 2 | 24 |
| 5 | 4 | 1 | 18 |

q →

| | i | j | v |
|---|---|---|---|
| 0 | 0 | 2 | -3 |
| 1 | 1 | 0 | 12 |
| 2 | | | |
| 3 | 2 | 0 | 9 |
| 4 | | | |
| 5 | | | |

col ↓

| col | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| num[col] | 1 | 2 | 2 | 0 | 0 | 1 |
| cpot[col] | 0 | 2 | 4 | 5 | 5 | 5 |

```
for (p = 0; p < M.tu; p ++) {
    col = M.data[p].j;     q   = cpot[col];
    T.data[q].i = M.data[p].j;
    T.data[q].j = M.data[p].i;
    T.data[q].e = M.data[p].e;
    cpot[col]++; }
```

| | i | j | v |
|---|---|---|---|
| 0 | 0 | 1 | 12 |
| 1 | 0 | 2 | 9 |
| p → 2 | 2 | 0 | -3 |
| 3 | 2 | 5 | 14 |
| 4 | 3 | 2 | 24 |
| 5 | 4 | 1 | 18 |

| | i | j | v |
|---|---|---|---|
| q → 0 | 0 | 2 | -3 |
| 1 | 1 | 0 | 12 |
| 2 | | | |
| 3 | 2 | 0 | 9 |
| 4 | | | |
| 5 | | | |

col ↓

| col | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| num[col] | 1 | 2 | 2 | 0 | 0 | 1 |
| cpot[col] | 1 | 2 | 4 | 5 | 5 | 5 |

```
for (p = 0; p < M.tu; p ++) {
    col = M.data[p].j;     q   = cpot[col];
    T.data[q].i = M.data[p].j;
    T.data[q].j = M.data[p].i;
    T.data[q].e = M.data[p].e;
    cpot[col]++; }
```

| | i | j | v |
|---|---|---|---|
| 0 | 0 | 1 | 12 |
| 1 | 0 | 2 | 9 |
| 2 | 2 | 0 | -3 |
| p → 3 | 2 | 5 | 14 |
| 4 | 3 | 2 | 24 |
| 5 | 4 | 1 | 18 |

| | i | j | v |
|---|---|---|---|
| 0 | 0 | 2 | -3 |
| 1 | 1 | 0 | 12 |
| 2 | | | |
| 3 | 2 | 0 | 9 |
| 4 | | | |
| q → 5 | 5 | 2 | 14 |

col ↓

| col | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| num[col] | 1 | 2 | 2 | 0 | 0 | 1 |
| cpot[col] | 1 | 2 | 4 | 5 | 5 | 5 |

```
for (p = 0; p < M.tu; p ++) {
    col = M.data[p].j;     q   = cpot[col];
    T.data[q].i = M.data[p].j;
    T.data[q].j = M.data[p].i;
    T.data[q].e = M.data[p].e;
    cpot[col]++; }
```

| | i | j | v |
|---|---|---|---|
| 0 | 0 | 1 | 12 |
| 1 | 0 | 2 | 9 |
| 2 | 2 | 0 | -3 |
| p → 3 | 2 | 5 | 14 |
| 4 | 3 | 2 | 24 |
| 5 | 4 | 1 | 18 |

| | i | j | v |
|---|---|---|---|
| 0 | 0 | 2 | -3 |
| 1 | 1 | 0 | 12 |
| 2 | | | |
| 3 | 2 | 0 | 9 |
| 4 | | | |
| q → 5 | 5 | 2 | 14 |

col ↓

| col | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| num[col] | 1 | 2 | 2 | 0 | 0 | 1 |
| cpot[col] | 1 | 2 | 4 | 5 | 5 | 6 |

# 本章小结

- 数组的类型定义
  - 重点掌握数组在内存中的存放规则
- 特殊矩阵的压缩
  - 对称矩阵、三角矩阵
  - 只需要保存上、下三角
  - 公式应该理解着记忆
- 稀疏矩阵的压缩
  - 三元组法
  - 稀疏矩阵的转置算法