CHAPTER 5

第5章 数组和稀疏矩阵

数组可以看成线性表的推广，二维数组称为矩阵，为了节省空间，可以对一些特殊矩阵和稀疏矩阵采用压缩存储方法。

本章的学习要点如下：

(1) 数组和一般线性表的差异。

(2) 数组的存储结构和元素地址计算方法。

(3) 各种特殊矩阵(如对称矩阵、上三角矩阵、下三角矩阵和对角矩阵)的压缩存储方法。

(4) 稀疏矩阵的各种存储结构以及基本运算实现算法。

(5) 灵活运用数组解决一些较复杂的应用问题。

视频讲解

5.1 数组

5.1.1 数组的基本概念

几乎所有的计算机语言都提供了数组类型，但直接将数组看成“连续的存储单元集合”是片面的，数组也分为逻辑结构和存储结构，尽管在计算机语言中实现数组通常采用连续的存储单元集合，但并不能说数组只能这样实现。

从逻辑结构上看，数组是二元组(idx，value)的集合，对每个 idx，都有一个 value 值与之对应，idx 称为下标，可以由一个整数、两个整数或多个整数构成，下标含有 $d(d\geqslant 1)$ 个整数称维数是 d。数组按维数分为一维、二维和多维数组。

一维数组 A 是 $n(n>1)$ 个相同类型元素 $a_0,a_1,a_2,\cdots,a_{n-1}$ 构成的有限序列，其逻辑表示为 $A=(a_0,a_1,a_2,\cdots,a_{n-1})$，其中，$A$ 是数组名，$a_i(0\leqslant i\leqslant n-1)$ 是数组 A 中序号为 i 的元素。

一个二维数组可以看作每个数据元素都是相同类型的一维数组的一维数组。以此类推，多维数组可以看作一个这样的线性表：其中的每个元素又

是一个线性表。

也可以这样看，一个 d 维数组中含有 $b_1 \times b_2 \times \cdots \times b_d$（假设第 i 维的大小为 b_i）个元素，每个元素受到 d 个关系的约束，且这 d 个关系都是线性关系。当 $d=1$ 时，数组就退化为定长的线性表；当 $d>1$ 时，d 维数组可以看成线性表的推广。例如，如图 5.1 所示的二维数组的逻辑关系用二元组表示如下：

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

图 5.1　一个二维数组

$B=(D,R)$
$R=\{r_1,r_2\}$
$r_1=\{<1,2>,<2,3>,<3,4>,<5,6>,<6,7>,<7,8>,<9,10>,<10,11>,<11,12>\}$
$r_2=\{<1,5>,<5,9>,<2,6>,<6,10>,<3,7>,<7,11>,<4,8>,<8,12>\}$

其中含有 12 个元素，这些元素之间有两种关系，r_1 表示行关系，r_2 表示列关系，r_1 和 r_2 均为线性关系。

一般地，数组具有以下特点：

① 数组中的各元素都具有统一的数据类型。

② $d(d\geqslant 1)$维数组中的非边界元素具有 d 个前驱元素和 d 个后继元素。

③ 数组的维数确定后，数据元素个数与元素之间的关系不再发生改变，特别适合于顺序存储。

④ 每个有意义的下标都存在一个与其相对应的数组元素值。

d 维数组抽象数据类型的定义如下：

ADT Array
{
数据对象：
　$D=\{$数组中的所有元素$\}$
数据关系：
　$R=\{r_1,r_2,\cdots,r_d\}$
　$r_i=\{$元素之间第 i 维的线性关系 $\mid i=1,2,\cdots,d\}$
基本运算：
　Value($A,i_1,i_2,\cdots,i_d$)：A 是已存在的 d 维数组，其运算结果是返回 $A[i_1,i_2,\cdots,i_d]$值.
　Assign($A,e,i_1,i_2,\cdots,i_d$)：A 是已存在的 d 维数组，其运算结果是置 $A[i_1,i_2,\cdots,i_d]=e$。
　…
}

数组的主要操作是存取元素值，如图 5.2 所示为一维数组的存取操作，由于数组没有插入和删除操作，所以数组通常采用顺序存储方式实现。

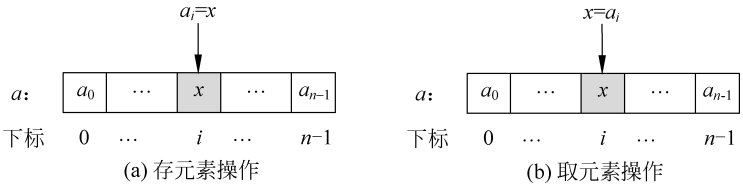


图 5.2　一维数组的基本操作

视频讲解

5.1.2 数组的存储结构

1. 一维数组

一维数组的所有元素依逻辑次序存放在一片连续的内存存储单元中,其起始地址为元素 a_0 的地址,即 LOC(a_0)。假设每个数据元素占用 k 个存储单元,则元素 a_i(0≤i<n)的存储地址 LOC(a_i)如下:

$$\mathrm{LOC}(a_i)=\mathrm{LOC}(a_0)+i\times k$$

该式说明一维数组中任一元素的存储地址可直接计算得到,即一维数组中的任一元素可直接存取,正因如此,一维数组具有随机存取特性。

C++中一维数组的定义方式如下:

```
T a[M];                                    //M 为常量,T 为数组元素类型
```

当然也可以使用 vector<T>容器作为一维动态数组。

2. d 维数组

对于 d(d≥2)维数组,必须约定其元素的存放次序(即存储方案),这是因为存储单元是一维的(计算机的存储结构是线性的),而数组是 d 维的。通常 d 维数组的存储方案有按行优先和按列优先两种。

下面以 m 行 n 列的二维数组 $A_{m\times n}=(a_{i,j})$为例进行讨论。

二维数组 A 按行优先存储的形式如图 5.3 所示,假设每个元素占用 k 个存储单元,用 LOC($a_{0,0}$)表示首元素 $a_{0,0}$ 的存储地址,则元素 $a_{i,j}$(0≤i,j<n)的存储地址如下:

$$\mathrm{LOC}(a_{i,j})=\mathrm{LOC}(a_{0,0})+(i\times n+j)\times k$$

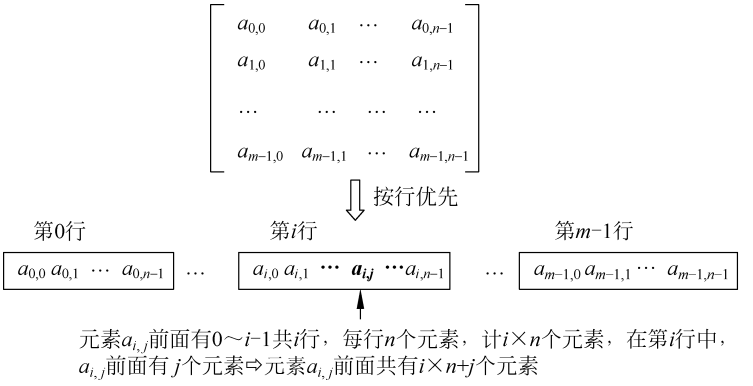


图 5.3 按行优先存储

二维数组 A 按列优先存储的形式如图 5.4 所示,在前面假设的情况下,元素 $a_{i,j}$ 的存储地址如下:

$$\mathrm{LOC}(a_{i,j})=\mathrm{LOC}(a_{0,0})+(j\times m+i)\times k$$

前面均假设二维数组的行、列下界为 0。更一般的情况是二维数组行下界是 c_1、行上界

是 d_1，列下界是 c_2、列上界是 d_2，即为数组 $A[c_1..d_1,c_2..d_2]$[①]，则该数组按行优先存储时有：

$$LOC(a_{i,j})=LOC(a_{c1,c2})+[(i-c_1)\times(d_2-c_2+1)+(j-c_2)]\times k$$

按列优先存储时有：

$$LOC(a_{i,j})=LOC(a_{c1,c2})+[(j-c_2)\times(d_1-c_1+1)+(i-c_1)]\times k$$

综上所述，从二维数组的元素地址计算公式 $LOC(a_{i,j})$ 看出，一旦数组元素的下标和元素类型确定，对应元素的存储地址就可以直接计算出来。也就是说，与一维数组相同，二维数组也具有随机存取特性。这样的推导公式和结论可推广至三维甚至更高维数组中。

C++中二维数组的定义方式如下：

```
T a[M][N];                                    //M、N 为常量，T 为数组元素类型
```

当然也可以使用 vector<vector<T>>容器作为二维动态数组。

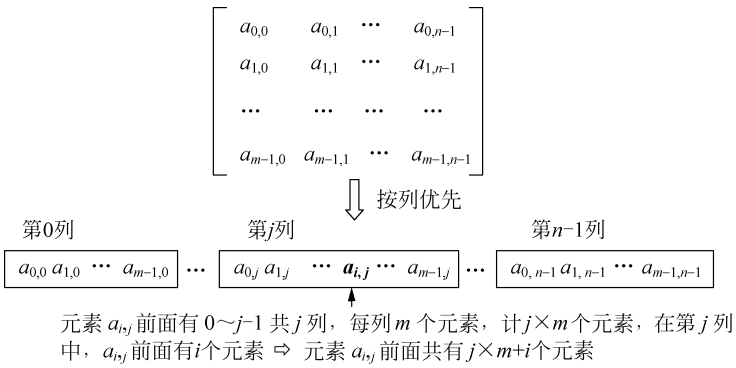


图 5.4　按列优先存储

【例 5.1】 设有二维数组 $a[1..50,1..80]$，其 $a[1][1]$ 元素的地址为 2000，每个元素占两个存储单元，若按行优先存储，则元素 $a[45][68]$ 的存储地址为多少？若按列优先存储，则元素 $a[45][68]$ 的存储地址为多少？

解：在按行优先存储时，元素 $a[45][68]$ 前面有 1～44 行，每行 80 个元素；计 44×80 个元素；在第 45 行中，元素 $a[45][68]$ 前面有 $a[45][1..67]$ 计 67 个元素，这样元素 $a[45][68]$ 前面存储的元素个数＝44×80＋67，所以 $LOC(a[45][68])=2000+(44\times80+67)\times2=9174$。

在按列优先存储时，元素 $a[45][68]$ 前面有 1～67 列，每列 50 个元素，计 67×50 个元素；在第 68 列中，元素 $a[45][68]$ 前面有 $a[1..44][68]$ 计 44 个元素，这样元素 $a[45][68]$ 前面存储的元素个数＝67×50＋44，所以 $LOC(a[45][68])=2000+(67\times50+44)\times2=8788$。

在 C++语言中二维及以上维的数组就是按行优先存储的，当在程序中采用数组存放大量的数据时，这些数据存放在内存中。当 CPU 读数组中的元素时并不是立即访问内存，而是先访问 Cache(高速缓存，其速度比访问内存快得多)，如果访问的数据在 Cache 中便直接

① $A[c_1..d_1,c_2..d_2]$ 表示数组 A 的行号从 c_1 到 c_2，列号从 d_1 到 d_2，通常数组下标从 0 开始，所以 $A[m]$ 数组可以表示为 $A[0..m-1]$，$A[m][n]$ 或 $A[m,n]$ 数组可以表示为 $A[0..m-1,0..n-1]$。

取相应的数据(称为命中),如果访问的数据不在 Cache 中才访问内存,并将访问数据所在的一个页块调入 Cache,如图 5.5 所示,这称为程序局部性原理。

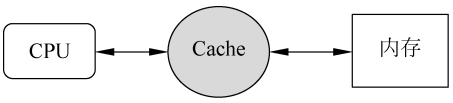


图 5.5　CPU 存取数据的方式

显然,如果程序中连续读取的数据在内存中是相邻存放的,那么 Cache 命中率高,程序执行速度快;反之,Cache 命中率低,程序执行速度慢。好的程序员可以利用程序局部性原理尽可能提高程序的性能。例如,有以下两个功能一模一样的程序:

```
//程序 A
int a[1000][50][8000];
int main()
{
    for (int i=0;i<1000;i++)
        for (int j=0;j<50;j++)
            for (int k=0;k<8000;k++)
                a[i][j][k]=i+j+k;
    return 0;
}
//程序 B
int a[1000][50][8000];
int main()
{
    for (int k=0;k<8000;k++)
        for (int j=0;j<50;j++)
            for (int i=0;i<1000;i++)
                a[i][j][k]=i+j+k;
    return 0;
}
```

程序 A 的执行时间为 1.597 秒,程序 B 的执行时间为 17.83 秒,相差 10 多倍。原因是数组 a 按行优先存放,而程序 A 恰好也是按行优先访问 a 中元素的,所以性能高;程序 B 则是按列优先访问 a 中元素的,所以性能低。

5.1.3　数组的应用

由于数组的使用简单、方便,特别是具有随机存取特性,因此在编程中数组被广泛地使用。使用数组的目的一方面是为了存储大量的数据类型相同的数据,避免重复性操作;另一方面用于模拟现实世界,例如顺序表就是采用数组模拟线性表。

下面的两个实战题分别是一维数组和二维数组的应用,前一个是有关前缀和,后一个是有关矩阵快速幂,前缀和与矩阵快速幂是高级编程中常用的技术。

【实战 5.1】　POJ2189——最多围栏个数


视频讲解

时间限制:1000ms;内存限制:65 536KB。

问题描述:一个条形牧场用 $p(1\leqslant p\leqslant 1000)$ 个柱子(编号分别为 1 到 p)分隔成等间距的围栏,每头牛只能在围栏中放牧,现有 $n(1\leqslant n\leqslant 1000)$ 头牛在放牧。请求出数量不超过

c(0≤c≤1000)头牛的最大连续区域的围栏个数。

输入格式：第 1 行是 3 个整数 n、p 和 c。第 2 行到第 $n+1$ 行，每行包含一个整数 x，取值范围为 1～$p-1$，指定一头牛在柱子 x 和柱子 $x+1$ 的围栏中放牧。注意一个围栏中允许放牧多头牛。

输出格式：在第 1 行中输出一个整数，表示最多有 c 头牛的最大连续区域的围栏个数。

【实战 5.2】 POJ3070——矩阵快速幂求 Fibonacci 数列

时间限制：1000ms；内存限制：65 536KB。


视频讲解

问题描述：Fibonacci 数列是 $F_0=0, F_1=1, F_n=F_{n-1}+F_{n-2}(n\geqslant 2)$。例如，前 10 项 Fibonacci 数列是 0,1,1,2,3,5,8,13,21,34。

Fibonacci 数列的另外一种写法是：

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \underbrace{\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}\cdots\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}}_{n\text{次}}$$

给定一个整数 n，请求出 F_n 的最后 4 位数字。

输入格式：输入包含多个测试用例，每个测试用例由单个包含整数 n(0≤n≤1 000 000 000)的行构成，以 $n=-1$ 表示结束。

输出格式：对于每个测试用例，输出一行包含 F_n 的最后 4 位数字，如果均为 0 则输出'0'，否则忽略前导 0(也就是输出 F_n mod 10 000)。


视频讲解

5.2　特殊矩阵的压缩存储

二维数组也称为矩阵。对于一个 m 行 n 列的矩阵，当 $m=n$ 时称为**方阵**，方阵的元素可以分为三部分，即上三角部分、主对角线部分和下三角部分，如图 5.6 所示。

所谓**特殊矩阵**，是指非零元素或常量元素的分布有一定规律的矩阵。为了节省存储空间，可以利用特殊矩阵的规律对它们进行压缩存储，例如让多个相同值的元素共享同一个存储单元等。这里主要讨论对称矩阵、三角矩阵和对角矩阵，它们都是方阵。

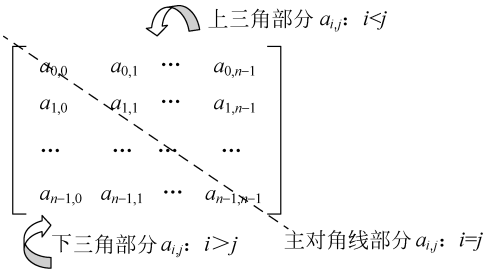


图 5.6　一个方阵的三部分

5.2.1 对称矩阵的压缩存储

若一个 n 阶方阵 $\boldsymbol{A}$ 的元素满足 $a_{i,j}=a_{j,i}(0\leqslant i,j\leqslant n-1)$，则称其为 n 阶**对称矩阵**。

如果直接采用二维数组存储对称矩阵，占用的内存空间为 n^2 个元素大小。由于对称矩阵的元素关于主对角线对称，因此在存储时可只存储其上三角和主对角线部分的元素，或者只存储其下三角和主对角线部分的元素，使得对称的元素共享同一存储空间，这样就可以将 n^2 个元素压缩存储到 $n(n+1)/2$ 个元素的空间中。不失一般性，在按行优先存储时仅存储其下三角和主对角线部分的元素，如图 5.7 所示。

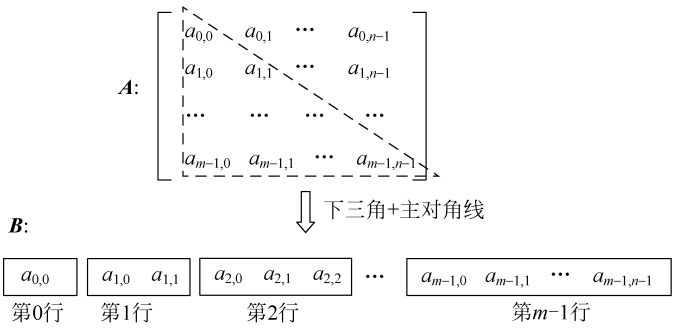


图 5.7 对称矩阵的压缩存储

采用一维数组 $\boldsymbol{B}=\{b_k\}$ 作为 n 阶对称矩阵 $\boldsymbol{A}$ 的压缩存储结构，在 $\boldsymbol{B}$ 中只存储对称矩阵 $\boldsymbol{A}$ 的下三角＋主对角线部分的元素 $a_{i,j}(i\geqslant j)$，这样 $\boldsymbol{B}$ 中的元素个数为 $n(n+1)/2$。

① 将 $\boldsymbol{A}$ 中下三角＋主对角线部分的元素 $a_{i,j}(i\geqslant j)$ 存储在 $\boldsymbol{B}$ 数组的 b_k 元素中。那么 k 和 i、j 之间是什么关系呢？

对于元素 $a_{i,j}$，求出它前面共存储的元素个数。不包括第 i 行，它前面共有 i 行(行下标为 0～$i-1$，第 0 行有 1 个元素，第 1 行有 2 个元素，…，第 $i-1$ 行有 i 个元素)，则这 i 行有 $1+2+\cdots+i=i(i+1)/2$ 个元素。在第 i 行中，元素 $a_{i,j}$ 前面的元素是 $a_{i,0}\sim a_{i,j-1}$，有 j 个元素，这样元素 $a_{i,j}$ 的前面共有 $i(i+1)/2+j$ 个元素，所以有 $k=i(i+1)/2+j$。也就是说，$\boldsymbol{A}$ 中下三角＋主对角线部分的元素 $a_{i,j}$ 存放在 $\boldsymbol{B}$ 中序号为 $i(i+1)/2+j$ 的元素中。

② 对于 $\boldsymbol{A}$ 中上三角部分的元素 $a_{i,j}(i<j)$，其值等于 $a_{j,i}$，而 $a_{j,i}$ 元素在 $\boldsymbol{B}$ 中的存储序号为 $j(j+1)/2+i$。

归纳起来，$\boldsymbol{A}$ 中任一元素 $a_{i,j}$ 和 $\boldsymbol{B}$ 中元素 b_k 之间存在如下对应关系：

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

上述关系用 $k=f(i,j)$ 表示，该函数是对称矩阵压缩存储的关键。所谓压缩存储，就是对于 n 阶对称矩阵 $\boldsymbol{A}$，采用一维数组 $\boldsymbol{B}$ 存储(几乎节省一半空间)，但需要提供类似 $\boldsymbol{A}[i][j]$ 元素的操作，实现过程如下：

① 存元素即执行 $\boldsymbol{A}[i][j]=x$，求出 $k=f(i,j)$，再执行 $\boldsymbol{B}[k]=x$。

② 取元素即执行 $y=\boldsymbol{A}[i][j]$，求出 $k=f(i,j)$，再执行 $y=\boldsymbol{B}[k]$。

说明：在计算对称矩阵的压缩存储关系函数 $k=f(i,j)$ 时需要考虑几点，压缩存储的

元素是上三角＋主对角线部分还是下三角＋主对角线部分？存储的元素是按行优先还是按列优先？矩阵的下标是从 1 开始还是从 0 开始？

【例 5.2】 有两个 n 阶整型对称矩阵 $\boldsymbol{A}$、$\boldsymbol{B}$。编写一个程序完成以下功能：

（1）将 $\boldsymbol{A}$、$\boldsymbol{B}$ 均采用按行优先顺序存放其下三角和主对角线部分元素的压缩存储方式，$\boldsymbol{A}$、$\boldsymbol{B}$ 的压缩结果存放在一维数组 a 和 b 中。

（2）通过 a 和 b 实现求 $\boldsymbol{A}$、$\boldsymbol{B}$ 的乘积运算，结果存放在二维数组 C 中。

要求采用相关数据进行测试。

解：对于两个 n 阶对称矩阵 $\boldsymbol{A}$、$\boldsymbol{B}$，在求乘积 C 数组时，计算公式如下。

$$C[i][j]=\sum_{k=0}^{n-1}\boldsymbol{A}[i][k]\times\boldsymbol{B}[k][j]$$

由于 $\boldsymbol{A}$、$\boldsymbol{B}$ 均采用 a、b 压缩存储，设计 getk(i,j)算法由 i、j 求压缩存储中的下标 k。在矩阵乘法中，求出 k1＝getk(i,k)，k2＝getk(k,j)，在求 $C[i][j]$ 时 $\boldsymbol{A}[i][k]$ 用 a[k1]替代，$\boldsymbol{B}[k][j]$ 用 b[k2]替代即可。

对应的程序如下：

```
#include<iostream>
#include<string>
using namespace std;
const int MAX=10;                          //最大维长度
void disp(int A[MAX][MAX],int n)           //输出 n 阶二维数组 A
{
    for (int i=0;i<n;i++)
    {
        for (int j=0;j<n;j++)
            printf("%3d",A[i][j]);
        printf("\n");
    }
}
void compression(int A[MAX][MAX],int n,int a[])   //将 A 压缩存储到 a 中
{
    for (int i=0;i<n;i++)
        for (int j=0;j<=i;j++)
        {
            int k=i*(i+1)/2+j;
            a[k]=A[i][j];
        }
}
int getk(int i,int j)                      //由 i,j 求压缩存储中的下标 k
{
    if (i>=j)
        return i*(i+1)/2+j;
    else
        return j*(j+1)/2+i;
}
void Mult(int a[],int b[],int C[MAX][MAX],int n)   //矩阵乘法
```

```
{
    for (int i=0;i<n;i++)
        for (int j=0;j<n;j++)
        {
            int s=0;
            for (int k=0;k<n;k++)
            {
                int k1=getk(i,k);
                int k2=getk(k,j);
                s+=a[k1]*b[k2];
            }
            C[i][j]=s;
        }
}
int main()
{
    int n=3;
    int m=n*(n+1)/2;
    int A[MAX][MAX]={{1,2,3},{2,4,5},{3,5,6}};
    int B[MAX][MAX]={{2,1,3},{1,5,2},{3,2,4}};
    int C[MAX][MAX];
    int* a=new int[m];
    int* b=new int[m];
    printf("A:\n"); disp(A,n);
    printf("A压缩存储到a中\n");
    compression(A,n,a);
    printf("a: ");
    for (int i=0;i<m;i++)
        printf("%3d",a[i]);
    printf("\n");
    printf("B:\n"); disp(B,n);
    printf("B压缩存储到b中\n");
    compression(B,n,b);
    printf("b: ");
    for (int i=0;i<m;i++)
        printf("%3d",b[i]);
    printf("\n");
    printf("C=A*B\n");
    Mult(a,b,C,n);
    printf("C:\n"); disp(C,n);
    return 0;
}
```

上述程序的执行结果如下：

```
A:
  1  2  3
  2  4  5
  3  5  6
```

```
A 压缩存储到 a 中
a:    1   2   4   3   5   6
B:
   2   1   3
   1   5   2
   3   2   4
B 压缩存储到 b 中
b:    2   1   5   3   2   4
C=A*B
C:
 13 17 19
 23 32 34
 29 40 43
```

5.2.2　三角矩阵的压缩存储


视频讲解

有些非对称的矩阵也可借用上述方法存储，例如 n 阶下(上)三角矩阵。所谓 n 阶**下(上)三角矩阵**，是指矩阵的上(下)三角部分(不包括主对角线)中的元素均为常数 c 的 n 阶方阵。

n 阶三角矩阵 $\boldsymbol{A}$ 采用一维数组 $\boldsymbol{B}=\{b_k\}$ 压缩存储，常量 c 仅存放一次，这样 $\boldsymbol{B}$ 中的元素个数为 $n(n+1)/2+1$。$\boldsymbol{A}$ 中任一元素 $a_{i,j}$ 和 $\boldsymbol{B}$ 中元素 b_k 之间存在着如下对应关系。

上三角矩阵：

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

下三角矩阵：

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

其中，B 的最后元素 $b_{n(n+1)/2}$ 中存放常数 c。

5.2.3　对角矩阵的压缩存储


视频讲解

若一个 n 阶方阵 $\boldsymbol{A}$ 满足其所有非零元素都集中在以主对角线为中心的带状区域中，其他元素均为 0，则称其为 n 阶**对角矩阵**。其主对角线上、下方各有 b 条次对角线，称 b 为矩阵半带宽，$(2b+1)$ 为矩阵的带宽。对于半带宽为 b $(0\leqslant b\leqslant(n-1)/2)$ 的对角矩阵，其 $|i-j|\leqslant b$ 的元素 $a_{i,j}$ 不为零，其余元素为零。图 5.8 所示为半带宽为 b 的对角矩阵示意图。

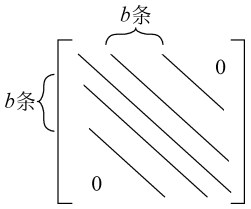


图 5.8　半带宽为 b 的对角矩阵

对于 $b=1$ 的三对角矩阵 $\boldsymbol{A}$，只存储其非零元素，并按行优先存储到一维数组 B 中，将 $\boldsymbol{A}$ 的非零元素 $a_{i,j}$ 存储到 B 的元素 b_k 中，k 的计算过程如下：

① 当 $i=0$ 时为第 0 行，共 2 个元素。

② 当 $i>0$ 时，第 0 行～第 $i-1$ 行共 $2+3(i-1)$ 个元素。

③ 对于非零元素 $a_{i,j}$，第 i 行最多 3 个元素，该行的首非零元素为 $a_{i,i-1}$（另外两个元素是 $a_{i,i}$ 和 $a_{i,i+1}$），即该行中元素 $a_{i,j}$ 前面存储的非零元素个数为 $j-(i-1)=j-i+1$。

所以，非零元素 $a_{i,j}$ 前面压缩存储的元素总个数 $=2+3(i-1)+j-i+1=2i+j$，即 $k=2i+j$。

以上讨论的对称矩阵、三角矩阵、对角矩阵的压缩存储方法是把有一定分布规律的值相同的元素（包括 0）压缩存储到一个存储空间中。这样的压缩存储只需在算法中按公式作一映射即可实现矩阵元素的随机存取。

5.3 稀疏矩阵

当一个阶数较大的矩阵中的非零元素个数 s 相对于矩阵元素的总个数 t 非常小时，即 $s \ll t$ 时，称该矩阵为**稀疏矩阵**。例如一个 100×100 的矩阵，若其中只有 100 个非零元素，就可称其为稀疏矩阵。

抽象数据类型稀疏矩阵与抽象数据类型 $d(d=2)$ 维数组的定义相似，这里不再介绍。


视频讲解

5.3.1 稀疏矩阵的三元组表示

由于稀疏矩阵中的非零元素个数很少，显然其压缩存储方法就是只存储非零元素。不同于前面介绍的各种特殊矩阵，稀疏矩阵中非零元素的分布没有规律（或者说随机分布），所以在存储非零元素时除了存储元素值还需存储对应的行、列下标。这样稀疏矩阵中的每一个非零元素需由一个三元组 $(i,j,a_{i,j})$ 表示，稀疏矩阵中的所有非零元素构成一个三元组线性表。

如图 5.9 所示为一个 6×7 阶稀疏矩阵 $\boldsymbol{A}$（为图示方便，所取的行、列数都很小）及其对应的三元组表示。从中看到，这里的稀疏矩阵三元组表示是一种顺序存储结构。

$$\boldsymbol{A}_{6\times7}=\begin{bmatrix}0&0&1&0&0&0&0\\0&2&0&0&0&0&0\\3&0&0&0&0&0&0\\0&0&0&5&0&0&0\\0&0&0&0&6&0&0\\0&0&0&0&0&7&4\end{bmatrix}$$

i	j	$a_{i,j}$
0	2	1
1	1	2
2	0	3
3	3	5
4	4	6
5	5	7
5	6	4

图 5.9 一个稀疏矩阵 A 及其对应的三元组表示

三元组表示中每个元素的类型定义如下：

```
struct TupElem                          //单个三元组元素的类型
{
    int r;                              //行号
    int c;                              //列号
    int d;                              //元素值
    TupElem() {}                        //构造函数
    TupElem(int r1,int c1,int d1)       //重载构造函数
```

```
    {
        r=r1;
        c=c1;
        d=d1;
    }
};
```

设计稀疏矩阵三元组存储结构类 TupClass 如下：

```
class TupClass                          //三元组存储结构类
{
    int rows;                           //行数
    int cols;                           //列数
    int nums;                           //非零元素个数
    TupElem * data;                     //稀疏矩阵对应的三元组顺序表
    //基本运算算法
};
```

TupClass 类中包含如下基本运算算法。

① CreateTup(**A**,m,n)：由 m 行 n 列的稀疏矩阵 **A** 创建其三元组表示。

② Setvalue(i,j,x)：利用三元组给稀疏矩阵的元素赋值，即执行 **A**[i][j]=x(x 为非零值)。

③ Getvalue(i,j,x)：利用三元组取稀疏矩阵的元素值即执行 x=**A**[i][j]。

④ DispTup()：输出稀疏矩阵的三元组表示。

其中，data 列表用于存放稀疏矩阵中的所有非零元素，通常按行优先顺序排列。这种有序性可简化大多数稀疏矩阵运算算法。

以 Setvalue(i,j,x)算法为例，其功能是将 **A**[i][j]赋值为一个非零值。实现过程是，第一步检查参数的正确性，当参数错误时返回 false。当参数正确时，第二步是查找，先按行号找到第 i 行的第一个非零元素，然后在第 i 行中查找第 j 列的元素(按有序性第 i 行的所有非零元素按列号顺序存放)。若找到了该元素，说明 **A**[i][j]本身是一个非零元素，只需要将其值替换为新值 x，否则说明 **A**[i][j]是一个零元素，需要修改为非零元素，此时在三元组中执行第三步，即插入操作，先将该位置后面的元素向后移动一个位置，然后在该位置存放 x 元素。对应的算法如下：

```
bool Setvalue(int i,int j,int x)        //三元组元素赋值:A[i][j]=x
{
    if (i<0 || i>=rows || j<0 || j>=cols)
        return false;                   //参数错误时返回 false
    int k=0,k1;
    while (k<nums && i>data[k].r)
        k++;                            //查找第 i 行的第一个非零元素
    while (k<nums && i==data[k].r && j>data[k].c)
        k++;                            //在第 i 行中查找第 j 列的元素
    if (data[k].r==i&& data[k].c==j)     //找到了这样的元素
        data[k].d=x;
    else                                //不存在这样的元素时插入一个元素
```

```
    {
        for (k1=nums-1; k1>=k;k1--)        //后移元素以便插入
        {
            data[k1+1].r=data[k1].r;
            data[k1+1].c=data[k1].c;
            data[k1+1].d=data[k1].d;
        }
        data[k].r=i; data[k].c=j; data[k].d=x;
        nums++;
    }
    return true;                                    //赋值成功时返回 true
}
```

说明：若稀疏矩阵采用一个二维数组存储，此时具有随机存取特性；若稀疏矩阵采用一个三元组顺序表存储，则不再具有随机存取特性。


视频讲解

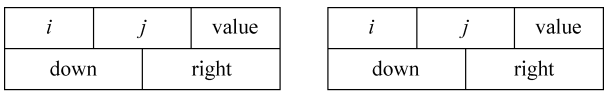
5.3.2 稀疏矩阵的十字链表表示

十字链表是稀疏矩阵的一种链式存储结构。

对于一个 $m \times n$ 的稀疏矩阵，每个非零元素用一个结点表示，该结点中存放该零元素的行号、列号和元素值。同一行中的所有非零元素结点链接成一个带行头结点的行循环单链表，同一列中的所有非零元素结点链接成一个带列头结点的列循环单链表。之所以采用循环单链表，是因为矩阵运算中经常是一行(列)操作完后进行下一行(列)操作，最后一行(列)操作完后进行首行(列)操作。

这样对稀疏矩阵的每个非零元素结点来说，它既是某个行链表中的一个结点，同时又是某个列链表中的一个结点，每个非零元素就好比在一个十字路口，由此称作**十字链表**。

每个非零元素结点的类型设计成如图 5.10(a)所示的结构，其中 i、j、value 分别代表非零元素所在的行号、列号和相应的元素值；down 和 right 分别称为向下指针和向右指针，分别用来链接同列中和同行中的下一个非零元素结点。这样行循环单链表个数为 m(每一行对应一个行循环单链表)，列循环单链表个数为 n(每一列对应一个列循环单链表)，那么行列头结点的个数就是 $m+n$。实际上，行头结点与列头结点是共享的，即 $h[i]$表示第 i 行循环单链表的头结点，同时也是第 i 列循环单链表的头结点，这里 $0 \leqslant i < \text{MAX}\{m,n\}$，即行列头结点的个数是 $\text{MAX}\{m,n\}$，所有行列头结点的类型与非零元素结点的类型相同。



(a) 元素结点类型　　(b) 头结点类型

图 5.10　十字链表结点结构

另外，将所有行列头结点再链接起来构成一个带头结点的循环单链表，这个头结点称为总头结点，即 hm，通过 hm 来标识整个十字链表。总头结点的类型设计成如图 5.10(b)所示的结构(之所以这样设计，是为了与非零元素结点的类型一致，这样在整个十字链表中采

用指针遍历所有结点时更方便),它的 link 域指向第一个行列头结点,其 i、j 域分别存放稀疏矩阵的行数 m 和列数 n,而 down 和 right 域没有作用。

从中看出,在 $m \times n$ 的稀疏矩阵的十字链表存储结构中有 m 个行循环单链表、n 个列循环单链表,另外有一个行列头结点构成的循环单链表,总的循环单链表个数是 $m+n+1$,总的头结点个数是 MAX$\{m,n\}+1$。

设稀疏矩阵如下:

$$\boldsymbol{B}_{3\times4}=\begin{bmatrix}1&0&0&2\\0&0&3&0\\0&0&0&4\end{bmatrix}$$

对应的十字链表如图 5.11 所示。为图示清楚,把每个行列头结点分别画成两个,实际上行、列值相同的头结点只有一个。

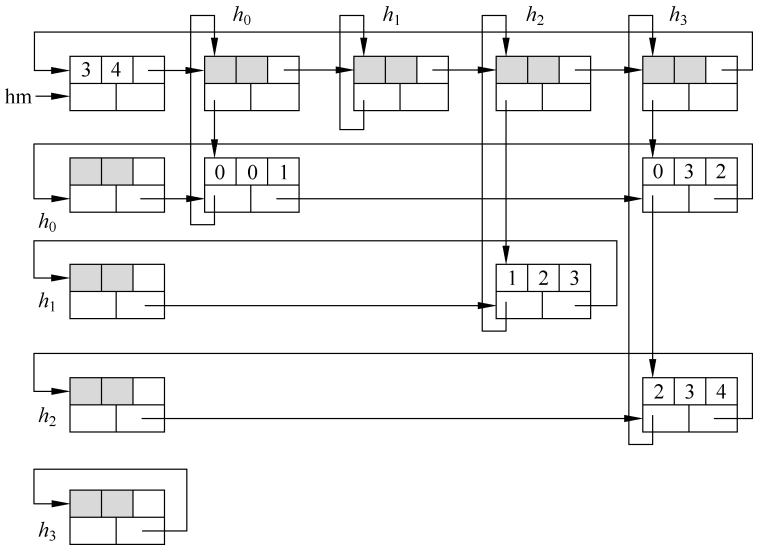


图 5.11　一个稀疏矩阵的十字链表

十字链表结点类型 MatNode 定义如下:

```
template < typename T >
struct MatNode                          //十字链表结点类型
{
    int row;                            //行号或者行数
    int col;                            //列号或者列数
    struct mtxn *  right, * down;       //行、列指针
    union
    {
        T value;                        //非零元素值
        struct mtxn *  link;            //指向下一个头结点
    } tag;
};
```

有关稀疏矩阵采用十字链表表示时的相关运算算法与三元组表示的类似,但更复杂,这里不再讨论。

【实战 5.3】 HDU4920——稀疏矩阵乘法


视频讲解

时间限制：2000ms；内存限制：131 072KB。

问题描述：给定两个 $n\times n$ 矩阵 $\boldsymbol{a}$ 和 $\boldsymbol{b}$，求它们的积，结果矩阵中每个元素值模 3。

输入格式：输入包含多个测试用例，每个测试用例的第一行为 $n(1\leqslant n\leqslant 800)$；接下来的 n 行每行 n 个整数，表示矩阵 $\boldsymbol{a}$，第 i 行的第 j 个整数为 $\boldsymbol{a}_{ij}$；类似地，再接下来的 n 行为矩阵 $\boldsymbol{b}(0\leqslant a_{ij},b_{ij}\leqslant 10^9)$。

输出格式：对于每个测试用例，输出 n 行，每行 n 个整数表示 $\boldsymbol{a}\times\boldsymbol{b}$ 的结果。

5.4 练习题

5.4.1 问答题

1. 为什么说数组是线性表的推广或扩展，而不说数组就是一种线性表？

2. 为什么数组一般不使用链式存储结构？

3. 如果某个一维整数数组 A 的元素个数 n 很大，存在大量重复的元素，且所有值相同的元素紧跟在一起，请设计一种压缩存储方式使得存储空间更节省。

4. 有一个 5×6 的二维数组 a，起始元素 $a[1][1]$的地址是 1000，每个元素的长度为 4。

(1) 采用按行优先存储，给出元素 $a[4][5]$的地址。

(2) 采用按列优先存储，给出元素 $a[4][5]$的地址。

5. 一个 n 阶对称矩阵存入内存，在采用压缩存储和非压缩存储时占用的内存空间分别是多少？

6. 一个 6 阶对称矩阵 $\boldsymbol{A}$ 中主对角线以上部分的元素已按列优先顺序存放于一维数组 B 中，主对角线上的元素均为 0。根据以下 B 的内容画出 $\boldsymbol{A}$ 矩阵。

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
B：	2	5	0	3	4	0	0	1	4	2	6	3	0	1	2

7. 设 $\boldsymbol{A}[0..9,0..9]$是一个 10 阶对称矩阵，采用按行优先将其下三角＋主对角线部分的元素压缩存储到一维数组 B 中。已知每个元素占用两个存储单元，其第一个元素 $\boldsymbol{A}[0][0]$的存储位置为 1000，求以下问题的计算过程及结果：

(1) 给出 $\boldsymbol{A}[4][5]$的存储位置。

(2) 给出存储位置为 1080 的元素的下标。

8. 设 n 阶下三角矩阵 $\boldsymbol{A}[0..n-1,0..n-1]$已压缩存储到一维数组 $B[1..m]$中，若按行为主序存储，则 $\boldsymbol{A}[i][j](i\geqslant j)$元素对应的 B 中存储位置为多少？给出推导过程。

9. 用十字链表表示一个有 k 个非零元素的 $m\times n$ 的稀疏矩阵，则其总的结点数为多少？

10. 特殊矩阵和稀疏矩阵哪一种压缩存储后失去随机存取特性？为什么？

5.4.2　算法设计题

1. 设计一个算法，将含有 n 个整数元素的数组 $a[0..n-1]$ 循环右移 m 位，要求算法的空间复杂度为 $O(1)$。

2. 有一个含有 n 个整数元素的数组 $a[0..n-1]$，设计一个算法求其中最后一个最小元素的下标。

3. 设 a 是一个含有 n 个元素的 double 型数组，b 是一个含有 n 个元素的整数数组，其值介于 0～$n-1$，且所有元素不相同。现设计一个算法，要求按 b 的内容调整 a 中元素的顺序，例如当 $b[2]=11$ 时，要求将 $a[11]$ 元素调整到 $a[2]$ 中。如 $n=5$，$a[]=\{1,2,3,4,5\}$，$b[]=\{2,3,4,1,0\}$，执行本算法后 $a[]=\{3,4,5,1,2\}$。

4. 设计一个算法，实现 m 行 n 列的二维数组 a 的就地转置，当 $m\neq n$ 时返回 false，否则返回 true。

5. 设计一个算法，求一个 m 行 n 列的二维整型数组 a 的左上角－右下角和右上角－左下角两条主对角线元素之和，当 $m\neq n$ 时返回 false，否则返回 true。

5.5　上机实验题

5.5.1　基础实验题

1. 编写一个实验程序，给定一个 m 行 n 列的二维数组 a，每个元素的长度 k，数组的起始地址 d，该数组按行优先还是按列优先存储，数组的初始下标 c1（假设 a 的行、列初始下标均为 c1），求元素 $a[i][j]$ 的地址，并用相关数据进行测试。

2. 编写一个实验程序，给定一个 n 阶对称矩阵 $\boldsymbol{A}$，采用压缩存储存储在一维数组 B 中，指出存储下三角＋主对角部分的元素还是上三角＋主对角部分的元素，按行优先还是按列优先，$\boldsymbol{A}$ 的初始下标 c1 和 B 的初始下标 c2，求元素 $a[i][j]$ 在 B 中的地址 k，并用相关数据进行测试。

3. 编写一个实验程序，假设稀疏矩阵采用三元组压缩存储，设计相关基本运算算法，并用相关数据进行测试。

5.5.2　应用实验题

1. 给定 $n(n\geqslant 1)$ 个整数的序列用整型数组 a 存储，要求求出其中的最大连续子序列之和。例如序列(－2,11,－4,13,－5,－2)的最大连续子序列和为 20，序列(－6,2,4,－7,5,3,2,－1,6,－9,10,－2)的最大连续子序列和为 16。分析算法的时间复杂度。

2. 求马鞍点问题。如果矩阵 a 中存在一个元素 $a[i][j]$ 满足条件“$a[i][j]$ 是第 i 行中值最小的元素，且又是第 j 列中值最大的元素”，则称之为该矩阵的一个马鞍点。设计一个程序，计算出 $m\times n$ 的矩阵 a 的所有马鞍点。

3. 对称矩阵压缩存储的恢复。一个 n 阶对称矩阵 $\boldsymbol{A}$ 采用一维数组 a 压缩存储，压缩方式是按行优先顺序存放 $\boldsymbol{A}$ 的下三角和主对角线部分的各元素。完成以下功能：

(1) 由 $\boldsymbol{A}$ 产生压缩存储 a。

(2) 由 b 恢复对称矩阵 $\boldsymbol{C}$。

通过相关数据进行测试。

5.6 在线编程题

1. LeetCode48——旋转图像
2. HDU1575——方阵 $\boldsymbol{A}$ 的迹
3. HDU1559——最大子矩阵
4. POJ3213——矩阵乘法问题
5. POJ3292——求 H 半素数个数