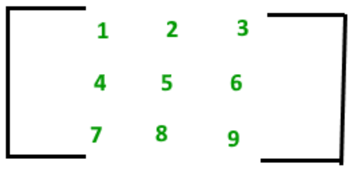
Introduction to Matrix

A **matrix** represents a collection of numbers arranged in order of rows and columns. It is necessary to enclose the elements of a matrix in parentheses or brackets.

A matrix with 9 elements is shown below:



The above Matrix M has 3 rows and 3 columns. Each element of matrix [M] can be referred to by its row and column number. For example, a_{23} = 6

Order of a Matrix : The order of a matrix is defined in terms of its number of rows and columns.

```
Order of a matrix = No. of rows × No. of columns

Therefore, Matrix [M] is a matrix of order 3 × 3.
```

Transpose of a Matrix

The transpose $[M]^T$ of an **m x n** matrix [M] is the n x m matrix obtained by interchanging the rows and columns of [M].

Transpose of a matrix A is defined as:

```
if A= [aij] mxn:
    then AT = [bij] nxm where bij = aji
```

For Example, transpose of matrix M, M^T will be:

```
MT  = 1 4 7
      2 5 8
      3 6 9
```

Properties of transpose of a matrix:

- $(A^T)^T = A$
- $(A+B)^T = A^T + B^T$
- $(AB)^T = B^T A^T$

Properties of Matrix addition and multiplication:

1. A+B = B+A (Commutative)
2. (A+B)+C = A+ (B+C) (Associative)
3. AB ≠ BA (Not Commutative)
4. (AB) C = A (BC) (Associative)
5. A (B+C) = AB+AC (Distributive)

Terminologies

- **Square Matrix:** A square Matrix has as many rows as it has columns. i.e. no of rows = no of columns.
- **Symmetric matrix:** A square matrix is said to be symmetric if the transpose of original matrix is equal to its original matrix. i.e. $(A^T) = A$.
- **Skew-symmetric:** A skew-symmetric (or antisymmetric or antimetric[1]) matrix is a square matrix whose transpose equals its negative.i.e. $(A^T) = -A$.
- **Diagonal Matrix:**A diagonal matrix is a matrix in which the entries outside the main diagonal are all zero. The term usually refers to square matrices.
- **Identity Matrix:**A square matrix in which all the elements of the principal diagonal are ones and all other elements are zeros.Identity matrix is denoted as I.
- **Orthogonal Matrix:** A matrix is said to be orthogonal if $AA^T = A^TA = I$.
- **Idemponent Matrix:** A matrix is said to be idemponent if $A^2 = A$.
- **Involutary Matrix:** A matrix is said to be Involutary if $A^2 = I$.
- **Singular Matrix**: A square matrix is said to be singular matrix if its determinant is zero i.e. |A|=0
- **Nonsingular Matrix**: A square matrix is said to be non-singular matrix if its determinant is non-zero.

Note: Every Square Matrix can uniquely be expressed as the sum of a symmetrix matrix and skew symmetric matrix. A = 1/2 (AT + A) + 1/2 (A - AT).

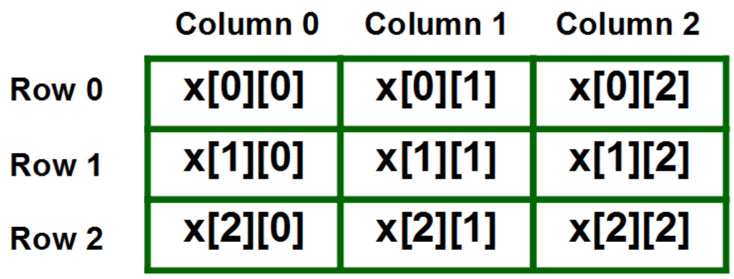
Trace of a matrix: trace of a matrix is denoted as tr(A) which is used only for square matrix and equals the sum of the diagonal elements of the matrix. For example:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \qquad tr(A) = 1+5+9 = 15$$

− Matrix Implementation

Matrix in programming languages can be implemented using **2-D arrays**. 2-D arrays or Two-Dimensional arrays in simple words can be defined as an *array of arrays*.

Elements in a 2-D array are stored in a tabular form in row major order. A two – dimensional array can be seen as a table with 'x' rows and 'y' columns where the row number ranges from 0 to (x-1) and column number ranges from 0 to (y-1). A two – dimensional array with 3 rows and 3 columns is shown below:



Declaring 2-D Arrays:

- Syntax for declaring 2-D Array in C++:

```
data_type  array_name[size1][size2]

Where,
data_type: Type of data to be stored in the array.
           Here data_type is valid C/C++ data type
array_name: Name of the array
size1: Number of rows
size2: Number of columns

Example:
int arr[2][5];

The above example, creates a 2-D array named arr with
2 rows and 5 columns in C/C++.
```

- Declaring 2-D array in Java:

```
data_type[][] array_name = new data_type[size1][size2]

Where,
data_type: Type of data to be stored in the array.
           Here data_type is valid Java data type
array_name: Name of the array
size1: Number of rows
size2: Number of columns

Example:
int arr[2][5];

The above example, creates a 2-D array named arr with
2 rows and 5 columns in Java.
```

Size of 2-D arrays: The total number of elements that can be stored in a 2-D array can be easily calculated by multiplying the size of both dimensions. For Example, the above declared array *arr* can store a maximum of 2*5 = 10 elements.

Accessing 2-D array elements

Elements in two-dimensional arrays are commonly referred by x[i][j] where '*i*' is the row number and '*j*' is the column number.

Syntax:

```
arr[row_index][column_index]
```

For example:

```
arr[0][0] = 1;
```

The above example represents the element present in first row and first column.

Note: In arrays if size of array is N. Its index will be from 0 to N-1. Therefore, for row_index 2, actual row number is 2+1 = 3.

Printing all elements of a 2-D array: To print all the elements of a Two-Dimensional array we can use nested for loops. We will require two for loops. One to traverse the rows and another to traverse columns.

Consider a 2-D array named **arr[][]** has **N** rows and **M** columns. Below code snippet traverses all of the elements of the 2-D array in row-major order and prints them:

```

// Traversing number of Rows
for (int i = 0; i < N; i++)
{
    // Traversing number of Columns
    for (int j = 0; j < M; j++)
    {
        // Access each element and print it
        print arr[i][j];
    }
}

```

Searching an element in a 2-D array: We can use a similar approach as above to search a given element if it is present in a 2-D array arr[] or not. The idea is to traverse the 2-D array using two nested loops and check for every element of the 2-D array if it matches with the given element. We will use a boolean flag, which will be set to true if the element is found in the 2-D array.

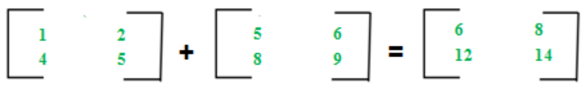
Consider a 2-D array named **arr[][]** has **N** rows and **M** columns. Below code snippet traverses all of the elements of the 2-D array in row-major order and check if the element **key** exists in it or not:

```

// Declare a boolean flag variable,
// initialized to false
boolean flag = false;

// Traversing number of Rows
for (int i = 0; i < N; i++)
{
    // Traversing number of Columns
    for (int j = 0; j < M; j++)
    {
        // Check if key is present
        // Set flag to true and stop
        // traversing further
        if(arr[i][j] == key)
        {
            flag = true;
            break;
        }
    }

    // If element found in the current row,
    // stop traversing further
    if(flag == true)
        break;
}

// The flag is now True if the element is present in
// the array otherwise it will be false.

```

- Matrix Operations (Addition, Subtraction, Multiplication)

Matrices Addition

The addition of two matrices A $_{m*n}$ and B$_{m*n}$ gives a matrix C$_{m*n}$. Here, m and n represents the number of rows and columns in the matrix respectively. The elements of C are sum of corresponding elements in A and B which can be shown as:



The algorithm for addition of matrices can be written as:

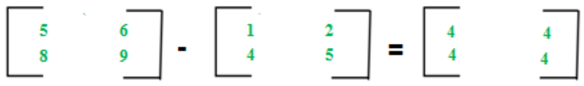
```
for i in 1 to m
   for j in 1 to n
      $c_{ij}$ = $a_{ij}$ + $b_{ij}$
```

Key points:

- Addition of matrices is commutative which means A+B = B+A
- Addition of matrices is associative which means A+(B+C) = (A+B)+C
- The order of matrices A, B and A+B is always same
- If order of A and B is different, A+B can't be computed
- The complexity of addition operation is O(m*n) where m*n is order of matrices

Matrices Subtraction

The subtraction of two matrices A_{m*n} and B_{m*n} gives a matrix C_{m*n}. Here, m and n represents the number of rows and columns in the matrix respectively. The elements of C are difference of corresponding elements in A and B which can be represented as:



The algorithm for subtraction of matrices can be written as:

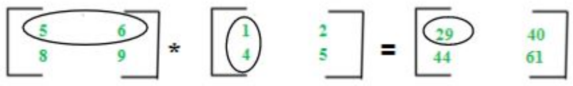
```
for i in 1 to m
   for j in 1 to n
      $c_{ij}$ = $a_{ij}$-$b_{ij}$
```

Key points:

- Subtraction of matrices is non-commutative which means A-B ≠ B-A
- Subtraction of matrices is non-associative which means A-(B-C) ≠ (A-B)-C
- The order of matrices A, B and A-B is always same
- If order of A and B is different, A-B can't be computed
- The complexity of subtraction operation is O(m*n) where m*n is order of matrices

Matrices Multiplication

The multiplication of two matrices A_{m*n} and B_{n*p} gives a matrix C_{m*p}. It means number of columns in A must be equal to number of rows in B to calculate C=A*B. To calculate element c_{11}, multiply elements of 1st row of A with 1st column of B and add them (5*1+6*4) which can be shown as:



The algorithm for multiplication of matrices A with order m*n and B with order n*p can be written as:

```
for i in 1 to m
   for j in 1 to p
      $c_{ij}$ = 0
      for k in 1 to n
         $c_{ij}$ += $a_{ik}$*$b_{kj}$
```

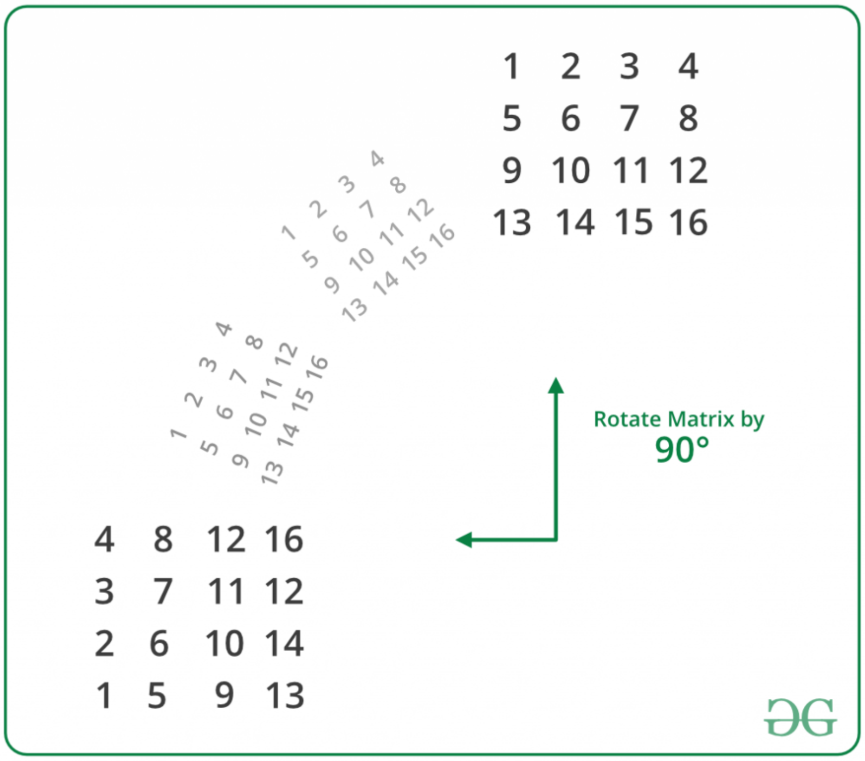
Key points:

- Multiplication of matrices is non-commutative which means A*B ≠ B*A
- Multiplication of matrices is associative which means A*(B*C) = (A*B)*C
- For computing A*B, the number of columns in A must be equal to number of rows in B
- Existence of A*B does not imply existence of B*A

- The complexity of multiplication operation (A*B) is O(m*n*p) where m*n and n*p are order of A and B respectively
- The order of matrix C computed as A*B is O(m*p) where m*n and n*p are order of A and B respectively

- Matrix Rotation

Given a **Square Matrix** of dimension **N * N**. The task is to rotate the matrix in anti-clock wise direction by 90 degrees.



On observing carefully, we can easily conclude that:

```
first row of destination ------> last column of source
second row of destination ------> second last column of source
.
.
.
.
last row of destination ------> first column of source
```

Therefore, rotating a matrix in anti-clockwise direction by 90 degrees is equivalent to replacing rows from top of the matrix by columns from the end.

Implementation of above approach: This method can be easily implemented by using extra space. The idea is to create a temporary matrix of same dimensions as that of the orginal matrix and copy the original matrix into this temporary matrix. Finally, replace each row in the orginal matrix one by one by columns of the temporary matrix from last to first.

Algorithm:

```
Original Matrix: mat[N][N].
Temporary Matrix: temp[N][N].

Copy original matrix into temporary matrix:
for(i = 0; i < N; i++)
{
    for(j = 0; j < N; j++)
    {
        temp[i][j] = mat[i][j];
    }
}

Updating Original Matrix by Rotated Matrix:
// Replace each row in the orginal matrix one by
// one by columns of the temporary matrix from
// last to first
for(i = 0; i < N; i++)
{
    for(j = 0; j < N; j++)
    {
        mat[i][j] = temp[j][N-i-1];
    }
}
```

Without Using Extra Space

The above problem can also be solved without using any addition matrix or extra-space. This is also called in-place rotating a square matrix by 90 degrees in anti-clockwise direction.

An N x N matrix will have floor(N/2) square cycles. For example, a 4 X 4 matrix will have 2 cycles. The first cycle is formed by its 1st row, last column, last row and 1st column. The second cycle is formed by 2nd row, second-last column, second-last row and 2nd column.

The idea is for each square cycle, we swap the elements involved with the corresponding cell in the matrix in anti-clockwise direction i.e. from top to left, left to bottom, bottom to right and from right to top one at a time. We use nothing but a temporary variable to achieve this.

Below steps demonstrate the idea:

```
First Cycle (Involves Red Elements)
 1  2  3 4
 5  6  7  8
 9 10 11 12
 13 14 15 16


Moving first group of four elements (First
elements of 1st row, last row, 1st column
and last column) of first cycle in counter
clockwise.
 4  2  3 16
 5  6  7 8
 9 10 11 12
 1 14  15 13

Moving next group of four elements of
first cycle in counter clockwise
 4  8  3 16
 5  6  7  15
 2  10 11 12
 1  14  9 13

Moving final group of four elements of
first cycle in counter clockwise
 4  8 12 16
 3  6  7 15
 2 10 11 14
 1  5  9 13


Second Cycle (Involves Blue Elements)
 4  8 12 16
 3  6 7  15
 2  10 11 14
 1  5  9 13

Fixing second cycle
 4  8 12 16
 3  7 11 15
 2  6 10 14
 1  5  9 13
```

Below function in-place rotates a square matrix by 90 degrees counter-clockwise:

```

// An Inplace function to rotate a N x N matrix
// by 90 degrees in anti-clockwise direction
void rotateMatrix(int mat[][N])
{
    // Consider all squares one by one
    for (int i = 0; i < N / 2; i++)
    {
        // Consider elements in group of 4 in
        // current square
        for (int j = i; j < N-i-1; j++)
        {
            // store current cell in temp variable
            int temp = mat[i][j];

            // move values from right to top
            mat[i][j] = mat[j][N-1-i];

            // move values from bottom to right
            mat[j][N-1-i] = mat[N-1-i][N-1-j];

            // move values from left to bottom
            mat[N-1-i][N-1-j] = mat[N-1-j][i];

            // assign temp to left
            mat[N-1-j][i] = temp;
        }
        }
    }
```

+ 2D Vector in C++

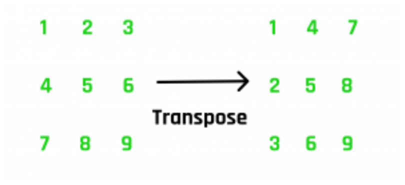
LIVE BATCHES

+ Implementing Matrix using 2D Arrays in Java

- Sample Problems on Matrix

Problem #1 : Transpose of a Matrix

Description - Transpose of a matrix is obtained by changing rows to columns and columns to rows. In other words, transpose of A[ ][ ] is obtained by changing A[i][j] to A[j][i].We are given a matrix of size m*n, We have to print the transpose of the matrix.



Solution : We have given matrix **A[m][n]**, We will create auxiliary matrix **B[n][m]** for storing the Transpose of the Matrix A. The idea is to place A [j][i] at B [i][j].
Pseudo Code

```
void transpose(A[m][n])
{
    B[n][m] // Transpose Matrix

    for ( i=0 to n-1 )
    {
        for ( j=0 to m-1 )
            B[i][j] = A[j][i]
    }
}
```

Time Complexity : O(m*n)
Auxiliary Space : O(m*n)

Problem #2 : Search Element in Row-wise and Column-wise Sorted Matrix

Description - Given an **n x n** matrix and a number **x**, find the **position of x** in the matrix. In the given matrix, every row and column is sorted in increasing order.
Solution : Idea is to solve problem with row and column elimination reducing the search space. Before jumping at the solution, lets try to understand the concept that is actually allowing us to solve the problem in linear time.
Let's start our search from the top-right corner of the array. There are three possible cases.

1. The number we are searching for is greater than the current number. This will ensure, that all the elements in the current row is smaller than the number we are searching for as we are already at the right-most element and the row is sorted. Thus, the entire row gets eliminated and we continue our search on the next row. Here elimination means we won't search on that row again.
2. The number we are searching for is smaller than the current number. This will ensure, that all the elements in the current column is greater than the number we are searching for. Thus, the entire column gets eliminated and we continue our search on the previous column i.e. the column at the immediate left.
3. The number we are searching for is equal to the current number. This will end our search.

Pseudo Code

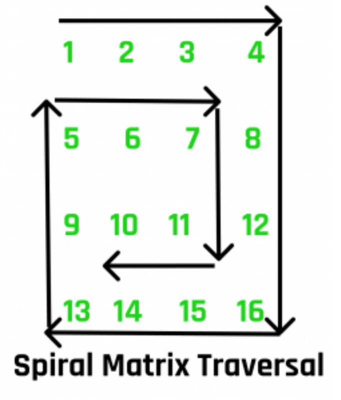
▲

```
// matrix size : n*n
void search(mat[][] ,int x)
{
    i = 0, j = n-1
    while(i > n && j >= 0 )
    {
        if (mat[i][j] == x )
        {
            print(i,j)
            break
        }
        else if (mat[i][j] > x )
        {
            j--
        }
        else
        {
            i++
        }
    }

}
```

Since, at each step, we are eliminating an entire row or column.
Time Complexity : O(n)
Auxiliary Space : O(1)

Problem #3 : Spiral Traversal of Matrix

Description - We are given a 2D Matrix of size **m*n**. We have to print the Matrix in Spiral form shown in the Example.



```
Output : 1 2 3 4 8 12 16 15 14 13 9 5 6 7 11 10
```

Solution : We will be traversing the Matrix in Spiral form with the help of 5 variables which includes iterator, starting and ending index of row and columns.

- k - starting row index
- m - ending row index
- l - starting column index
- n - ending column index
- i - iterator

Pseudo Code

```
void print_spiral(A[][], m, n)
{
    k = 0, l = 0
    while (k < m && l < n)
    {
        /* Print the first row from the remaining rows */
        for (i=l to n-1)
            print(A[k][i])
        k++

        /* Print the last column from the remaining columns */
        for (i = k to i = m-1 )
            print(A[i][n-1])
        n--

        /* Print the last row from the remaining rows */
        if ( k < m)
        {
            for (i = n-1; i >= l; --i)
                print(A[m-1][i])
            m--
        }

        /* Print the first column from the remaining columns */
        if (l < n) :
        {
            for (i = m-1; i >= k; --i)
                print(A[i][l])
            l++
        }
    }
}
```

- Determinant of a Matrix | Explanation

What is the Determinant of a Matrix?

Determinant of a Matrix is a special number that is defined only for square matrices (matrices which have the same number of rows and columns). The determinant is used at many places in calculus and other concepts related to algebra, it actually represents the matrix in term of a real number which can be used in solving a system of linear equations and finding the inverse of a matrix.

How to calculate Determinant?

The value of the determinant of a matrix can be calculated by the following procedure:

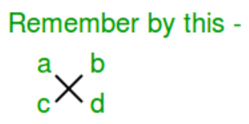
> *For each element of the first row or first column get **cofactor** of those elements and then multiply the element with the determinant of the corresponding cofactor, and finally add them with alternate signs. As a base case, the value of the determinant of a 1*1 matrix is the single value itself.*

Cofactor of an element, is a matrix which we can get by removing row and column of that element from that matrix.

Examples

Determinant of 2 x 2 Matrix:

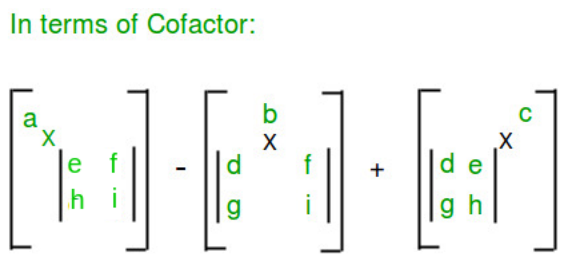
```
A = | a   b |
    | c   d |

|A| = (ad - bc)
```



(https://media.geeksforgeeks.org/wp-content/uploads/matrixcross.png)

Determinant of 3 x 3 Matrix:

```
    | a b c |
A = | d e f |
    | g h i |

|A| = a(ei - fh) - b(di - gf) + c(dh - eg)
```



Calculating Determinant of an NxN Matrix

To calculate determinant of an NxN matrix, we will create two functions namely:

- **determinantOfMatrix()**: This function will accept two parameters, the matrix and its dimension and will return the value of the determinant of the given matrix.
- **getCofactor()**: This function will return cofactor of the given element in the first row of a given matrix.

The step-by-step approach is:

- Inside the **determinantOfMatrix()** function, run a loop from 0 to N for all elements of the first row.
- For every element in the first row, find its cofactor matrix using the **getCofactor()** function.
- Then, multiply the element with the determinant of the corresponding cofactor, and finally add them with alternate signs.
- To find the determinant of the current co-factor, call the determinantOfMatrix() function recursively, and pass the cofactor matrix and its dimension as parameters.

Implementation of the getDeterminant() function:

```

/* Recursive function for finding determinant of matrix.
   n is current dimension of mat[][]. */
int determinantOfMatrix(int mat[N][N], int n)
{
    int D = 0; // Initialize result

    //  Base case : if matrix contains single element
    if (n == 1)
        return mat[0][0];

    int temp[N][N]; // To store cofactors

    int sign = 1;  // To store sign multiplier

     // Iterate for each element of first row
    for (int f = 0; f < n; f++)
    {
        // Getting Cofactor of mat[0][f]
        getCofactor(mat, temp, 0, f, n);
        D += sign * mat[0][f] * determinantOfMatrix(temp, n - 1);

        // terms are to be added with alternate sign
        sign = -sign;
    }

    return D;
}
```

Implementation of the getCofactor() function:

```

// Function to get cofactor of mat[p][q] in temp[][]. n is current
// dimension of mat[][]
void getCofactor(int mat[N][N], int temp[N][N], int p, int q, int n)
{
    int i = 0, j = 0;

    // Looping for each element of the matrix
    for (int row = 0; row < n; row++)
    {
        for (int col = 0; col < n; col++)
        {
            //  Copying into temporary matrix only those element
            //  which are not in given row and column
            if (row != p && col != q)
            {
                temp[i][j++] = mat[row][col];

                // Row is filled, so increase row index and
                // reset col index
                if (j == n - 1)
                {
                    j = 0;
                    i++;
                }
            }
        }
    }
}

```