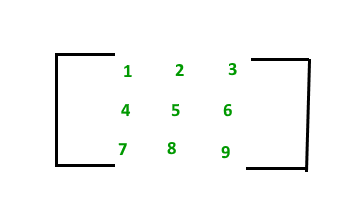
**Topic 1 : Introduction to Matrix**

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, a23 = 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, MT will be:

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

**Properties of transpose of a matrix:**

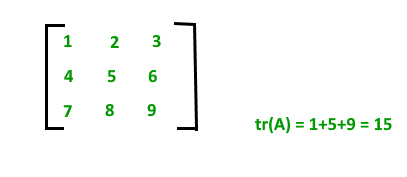
* (AT)T = A
* (A+B)T = AT + BT
* (AB)T = BTAT

**Properties of Matrix addition and multiplication:**

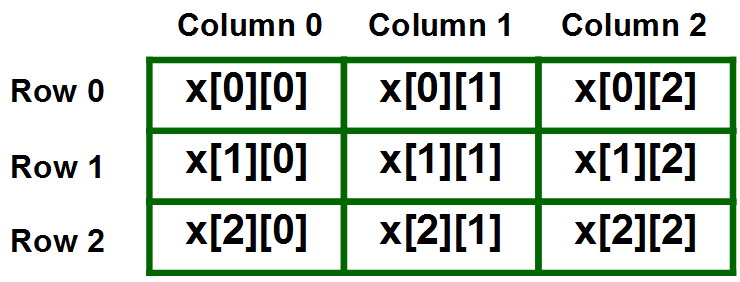
1. A+B = B+A (Commutative)
2. (A+B)+C = A+ (B+C) (Associative)
3. AB ≠ BA (Not Commutative)
4. (AB) C = A (BC) (Associative)
5. A (B+C) = AB+AC (Distributive)

**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 the original matrix is equal to its original matrix. i.e. (AT) = A.
* **Skew-symmetric:** A skew-symmetric (or antisymmetric or antimetric[1]) matrix is a square matrix whose transpose equals its negative.i.e. (AT) = -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 AAT = ATA = I.
* **Idempotent Matrix:** A matrix is said to be idempotent if A2 = A.
* **Involutory Matrix:** A matrix is said to be Involuntary if A2 = 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 a non-singular matrix if its determinant is non-zero.

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


**Topic 2: 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 to 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[][]** that 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:



1

2

3

4

5

6

7

8

9

10

11

12

// 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[][]** that 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:



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

// 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;

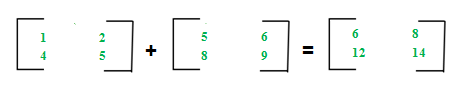
}

// The flag is now True if the element is present in

// the array otherwise it will be false.

**Topic 3 : Matrix Operations**

**Matrices Addition**

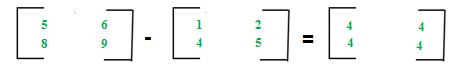
The addition of two matrices A m\*n and Bm\*n gives a matrix Cm\*n. Here, m and n represents the number of rows and columns in the matrix respectively. The elements of C are sum of corresponding elements in A and B which can be shown as:  
  
  
  
The algorithm for addition of matrices can be written as:

for i in 1 to m  
 for j in 1 to n  
 cij = aij + bij

**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**

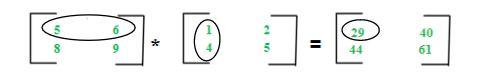
The subtraction of two matrices Am\*n and Bm\*n gives a matrix Cm\*n. Here, m and n represents the number of rows and columns in the matrix respectively. The elements of C are difference of corresponding elements in A and B which can be represented as:  
  
  
  
The algorithm for subtraction of matrices can be written as:

for i in 1 to m  
 for j in 1 to n  
 cij = aij-bij

**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**

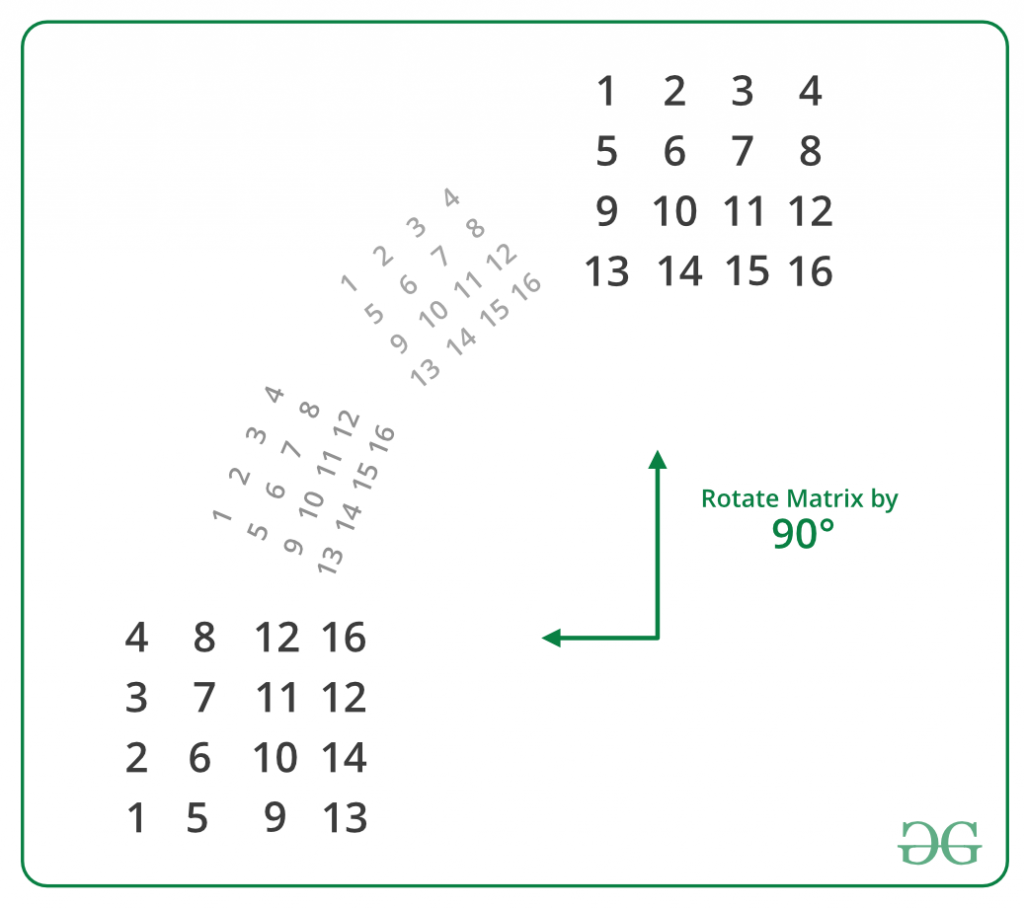
The multiplication of two matrices Am\*n and Bn\*p gives a matrix Cm\*p. It means number of columns in A must be equal to number of rows in B to calculate C=A\*B. To calculate element c11, multiply elements of 1st row of A with 1st column of B and add them (5\*1+6\*4) which can be shown as:  
  
  
  
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  
 cij = 0  
 for k in 1 to n  
 cij += aik\*bkj

**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

**Topic 4 : Matrix Rotation**

Given a **Square Matrix** of dimension **N \* N**. The task is to rotate the matrix in anticlockwise 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 an 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 the same dimensions as that of the original matrix and copy the original matrix into this temporary matrix. Finally, replace each row in the original 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 original 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 additional matrix or extra-space. This is also called in-place rotating a square matrix by 90 degrees in an 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 an 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 counterclockwise:

**Topic 5 : 2D Vector in C++**

Matrix in C++ can be implemented using 2D arrays or Vectors. Just like arrays, 2D vectors means **vector of vector**.  
  
**Normal Vector Declaration:**

vector< data\_type > vec\_name;

**2-D Vector Declaration:**

vector< vector < data\_type > > vec\_name;

**Calculating the number of rows and columns**:

* The number of rows in a 2D Vector can be found by calculating the size of the outer vector as *vec\_name.size()*.
* The number of items in each row of a 2D Vector can be found by calculating the size of each row as *vec\_name[i].size()*.

Below program illustrate 2D vectors by declaring and printing all elements of a 2D vector:



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

// C++ code to demonstrate 2D vector

#include <iostream>

#include <vector> // for 2D vector

using namespace std;

int main()

{

// Initializing 2D vector "vect" with

// values

vector<vector<int> > vect{ { 1, 2, 3 },

{ 4, 5, 6 },

{ 7, 8, 9 } };

// Displaying the 2D vector

for (int i = 0; i < vect.size(); i++) {

for (int j = 0; j < vect[i].size(); j++)

cout << vect[i][j] << " ";

cout << endl;

}

return 0;

}

Run

**Output**:

1 2 3   
4 5 6   
7 8 9

**Note**: The functions of the vector can be used with 2D vectors as well.

**Topic 6 : Implementing Matrix Using 2D Arrays in Java**

Two - dimensional array is the simplest form of a multidimensional array. A two - dimensional array can be seen as an array of one - dimensional array for easier understanding.  
  
**Indirect Method of Declaration:**

* **Declaration - Syntax:**
* **data\_type[][] array\_name = new data\_type[x][y];**
* For example: int[][] arr = new int[10][20];

* **Initialization - Syntax:**
* **array\_name[row\_index][column\_index] = value;**
* For example: arr[0][0] = 1;

**Example:**



1

2

3

4

5

6

7

8

9

10

11

12

13

class GFG {

public static void main(String[] args)

{

int[][] arr = new int[10][20];

arr[0][0] = 1;

System.out.println("arr[0][0] = " + arr[0][0]);

}

}

Run

**Output:**

arr[0][0] = 1

**Direct Method of Declaration:**  
**Syntax:**

**data\_type[][] array\_name = {  
 {valueR1C1, valueR1C2, ....},   
 {valueR2C1, valueR2C2, ....}  
 };**  
For example: int[][] arr = {{1, 2}, {3, 4}};

**Example:**



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

class GFG {

public static void main(String[] args)

{

int[][] arr = { { 1, 2 }, { 3, 4 } };

for (int i = 0; i < 2; i++)

for (int j = 0; j < 2; j++)

System.out.println("arr[" + i + "][" + j + "] = "

+ arr[i][j]);

}

}

Run

**Output:**

arr[0][0] = 1  
arr[0][1] = 2  
arr[1][0] = 3  
arr[1][1] = 4

Accessing Elements of Two-Dimensional Arrays

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:**

x[row\_index][column\_index]

For example:

int[][] arr = new int[10][20];

arr[0][0] = 1;

The above example represents the element present in the 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.  
  
**Example:**



1

2

3

4

5

6

7

8

9

10

11

12

class GFG {

public static void main(String[] args)

{

int[][] arr = { { 1, 2 }, { 3, 4 } };

System.out.println("arr[0][0] = " + arr[0][0]);

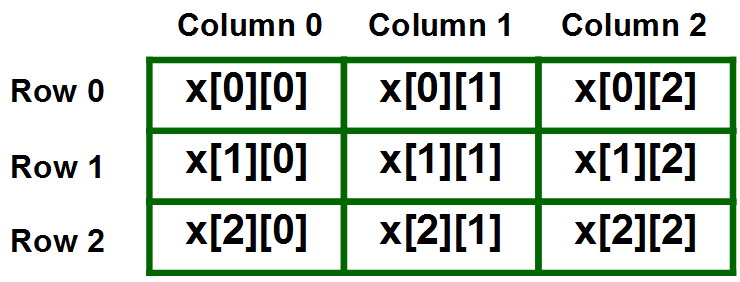
}

}

Run

**Output:**

arr[0][0] = 1

**Representation of 2D array in Tabular Format:** 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 'x' with 3 rows and 3 columns is shown below:  
  
  
  
**Print 2D array in tabular format:**  
To output all the elements of a Two-Dimensional array, use nested for loops. For this two for loops are required, One to traverse the rows and another to traverse columns.  
  
**Example:**



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

class GFG {

public static void main(String[] args)

{

int[][] arr = { { 1, 2 }, { 3, 4 } };

for (int i = 0; i < 2; i++) {

for (int j = 0; j < 2; j++) {

System.out.print(arr[i][j] + " ");

}

System.out.println();

}

}

}

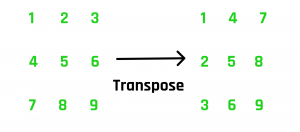
Run

**Output:**

1 2   
3 4

**Topic 7 : 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 the matrix **A[ m ][ n ]**, We will create an 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 problems with row and column elimination reducing the search space. Before jumping at the solution, let's 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.

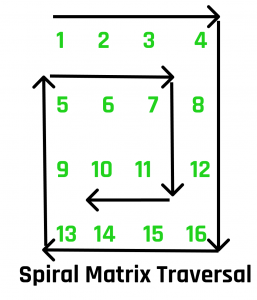
1. The number we are searching for is greater than the current number. This will ensure that all the elements in the current row are smaller than the number we are searching for as we are already at the right-most element and the row is sorted. Thus, the entire row gets eliminated and we continue our 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 are 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 the 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++  
 }  
 }  
}