INTRODUCTION TO 2D-ARRAY

CODE VIEW

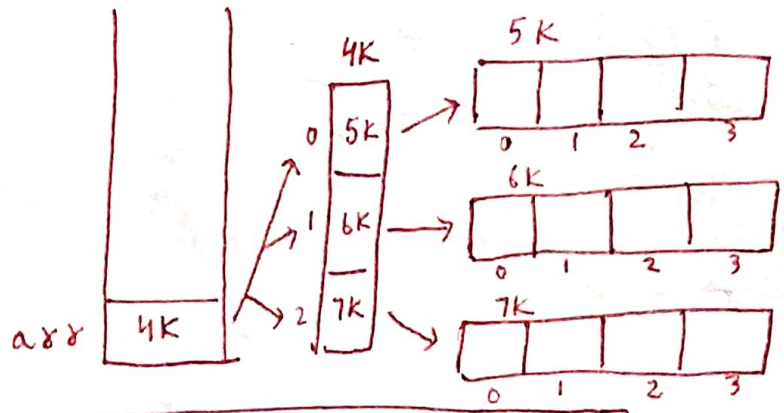
```
pc m{
p s v m(s[]a){
int [][]arr = new int[3][4];
a[0][0] = 11;
   [1] = 12;
   [2] = 13;
   [3] = 14;
arr[1][0] = 21;
   [1] = 22;
   [2] = 23;
   [3] = 24;
arr[2][0] = 31;
   [1] = 32;
   [2] = 33;
   [3] = 34;
for(int i=0;i<arr.length;i++)
{
for(int j=0;j<arr[0].length;j++)
{ Syso(arr[i][j]+"\t");
 }
 Syso();
 }
```

ABSTRACT VIEW

	0	1	2	3
0	0,0	0,1	0,2	0,3
1	1,0	1,1	1,2	1,3
2	2,0	2,1	2,2	2,3

11	12	13	14
21	22	23	24
31	32	33	34

MEMORY VIEW



NO. OF ROWS = arr.length
NO. OF COLUMNS = arr[0].length

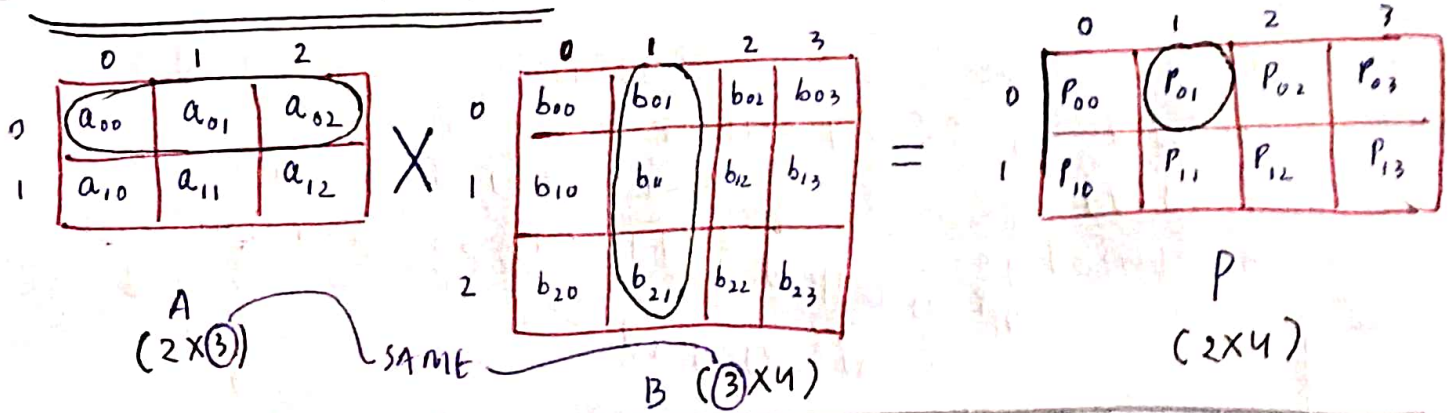
```
Scanner s=new Scanner(System.in);
 int n = s.nextInt();
 int m = s.nextInt();
 int [][]arr = new int[n][m];
 for(int i = 0; i<n; i++){
  for(int j = 0; j<m; j++){
      arr[i][j] = s.nextInt();
   }
 }
```

→ Code For 2D ARRAY Demo

Question

INTRODUCTION TO 2D-ARRAY

MATRIX MULTIPLICATION



$$P_{ij} = a_{i0}b_{0j} + a_{i1}b_{1j} + \ldots + a_{i(n-1)}b_{(n-1)j}$$

$$= \sum_{K=0}^{n-1} a_{iK}b_{Kj}$$

$$\therefore P_{01} = a_{00} * b_{01} + a_{01} * b_{11} + a_{02} * b_{21}$$

= EK SPOT FILL KRNE KE LIYE,
JO ISS SPOT KI ROW HAI WOH REHTI HAI 1ST MATRIX KI ROW.
JO ISS SPOT KA COLUMN HAI WOH REHTA HAI 2ND MATRIX KA COLUMN.
→ ISSME COLUMNS CHANGE HOTY HAI!
→ ISSME ROWS CHANGE HOTI HAI!

= (a) me ROW SAME HAI; COLUMNS CHANGE HUAY!
= (b) me ROW CHANGE HUI; COLUMNS SAME RAHEY!

APPROACH

(1) PHELE 1ST MATRIX KA INPUT LOO.. → row, column, array
(2) FIR 2ND MATRIX KA INPUT LOO... → row, column, array
(3) CONDITION FOR INVALID INPUT → `C1 != R2`
(4) NOW, PRODUCT KI MATRIX BANENI! → `int[][] prd = new int[r1][c2]`
(5) LOOP CHALENA PRODUCT MATRIX PE!
  → PHELA LOOP → matrix banake print krne ke liye!
  → DUSRA LOOP → print krane ke liye!

MULTIPLICATION OF 2D-ARRAY

```
p s v m(S[]a){
Scanner s= new Scanner (System.in);
int r1 = s.nextInt(); → ROW
int c1 = s.nextInt(); → COLUMN
int[][] arr1 = new int[r1][c1];
for(int i=0; i<r1; i++){
  for(int j=0; j<c1; j++){
      arr1[i][j] = s.nextInt();
  }
}
```

FIRST ARRAY

Time complexity

$O(n^3)$

3 nested loops are used. Cubic

```
int r2 = s.nextInt(); → ROW
int c2 = s.nextInt(); → COLUMN
int [][]arr2 = new int [r2][c2];
for(int i=0; i<r2; i++){
  for(int j=0; j<c2; j++){
    arr2[i][j] = s.nextInt();
  }
}
```

SECOND ARRAY

SPACE COMPLEXITY

$O(n^2)$

As 2D Arrays are used to store numbers.
∴ space complexity is Quadratic

```
if(c1 != r2){
  System.out.print("INVALID INPUT");
  return;
}
```

CONDITION FOR INVALID INPUT

```
int[][] prd = new int[r1][c2];
for(int i=0; i<r1; i++){          → prd.length
  for(int j=0; j<c2; j++){        → prd[0].length
    for(int k=0; k<r1; k++){
      prd[i][j] += (arr1[i][k] * arr2[k][j]);
    }
  }
}
for(int i=0; i<r1; i++){          → prd.length
  for(int j=0; j<c2; j++){        → prd[0].length
    System.out.print(prd[i][j]+" ");
  } System.out.println();
}}
}
```

NEW PRODUCT MATRIX ARRAY CREATION

PRINTING PRODUCT MATRIX

⑧ THE STATE OF WAKANDA - 1

The historic state of wakanda has various monuments and souvenirs which are visited by many travelers every day. The guides follow a prescribed route of visiting the monuments which improve their understanting of the relevance of each monument.

The route of the monument is fixed and expressed in a 2-d matrix where the travelers visit the prescribed next monument.

For example:

1	2	3
4	5	6
7	8	9

∴ The prescribed route and the visitors travel this path:

$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9$

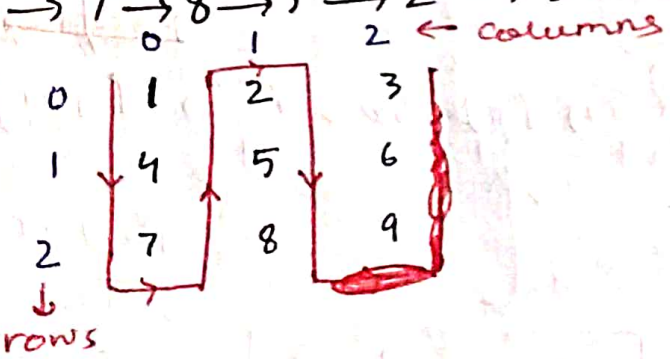
However, a certain visitor decides to travel a different path as follows:

1. He First travels southwards till no further south places are available.
2. He then moves only 1 place east wards.
3. He starts to move again towards north till any further north moves are available.

This continues till all the places are covered.

Path followed by this travelers:

$1 \rightarrow 4 \rightarrow 7 \rightarrow 8 \rightarrow 5 \rightarrow 2 \rightarrow 3 \rightarrow 6 \rightarrow 9$



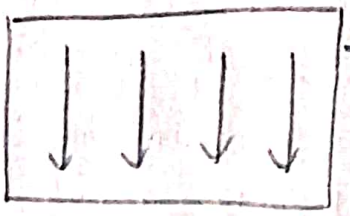
WAVE DISPLAY / TRAVERSAL

```
p s v m(s[] a){
 Scanner s= new Scanner(System.in);
 int r=s.nextInt();
 int c= s.nextInt();
 int [][]arr= new int[r][c];
for(int i=0; i<arr.length; i++){
    for(int j=0; j<arr[0].length; j++){
        arr[i][j] = s.nextInt();
    }
 }
```

```
for (int j=0; j<arr[0].length; j++) {
    for (int i=0; i<arr.length; i++) {
        System.out.print(arr[i][j]);
    }
}
```

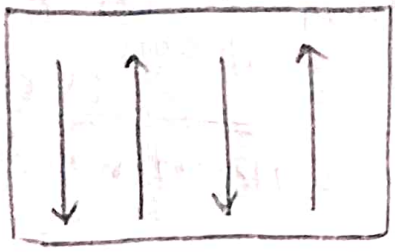
	j=0	1	2	3
i=0 →	11	12	13	14
1 →	21	22	23	24
2 →	31	32	33	34

OUTPUT



11
21
31
12
22
32
13
23
33
14
24
34

we need



EVEN-COLUMNS ← niche ki trafh

ODD-COLUMNS ↓ upper ki trafh

Time Complexity

$O(n^2)$

As there is nested loop and the outer for loop runs (n) times. There are two inner for loops.

making the time Complexity:

$O(n) * (O(n/2) + O(n/2)) = O(n^2)$

Space Complexity

$O(1)$

```
for (int j=0; j<arr[0].length; j++) {
    if (j%2==0) { //EVEN
        for (int i=0; i<arr.length; i++) {
            System.out.print(arr[i][j]);
        }
    }
    else { //ODD
        for (int i=arr.length-1; i>=0; i--) {
            System.out.print(arr[i][j]);
        }
    }
}
```

// Bhar columns ka loop hai!
// Ander rows ka loop!
// Agar even columns hai toh sidha loop print hoga!
// Agar odd columns hai toh ulta loop print hoga!
// Hamesha last vala element (arr.length-1) hota hai!

⑧ SPIRAL DISPLAY :

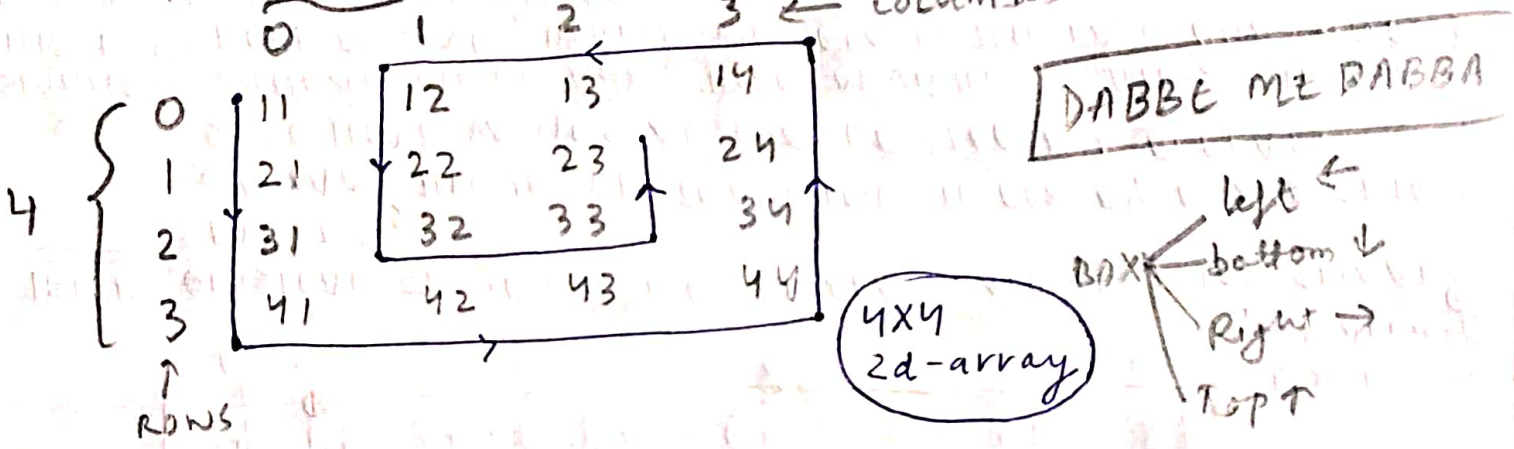
YOU WILL BE GIVEN A NUMBER (n), REPRESENTING THE NUMBER OF ROWS, THEN A NUMBER (m), REPRESENTING THE NUMBER OF COLUMNS. AFTER THAT WE ~~[illegible]~~ WILL BE GIVEN (n*m) numbers, REPRESENTING 2D ARRAY.

WE NEED TO TRAVERSE THIS INPUT 2D ARRAY, IN A SPIRAL PATTERN.

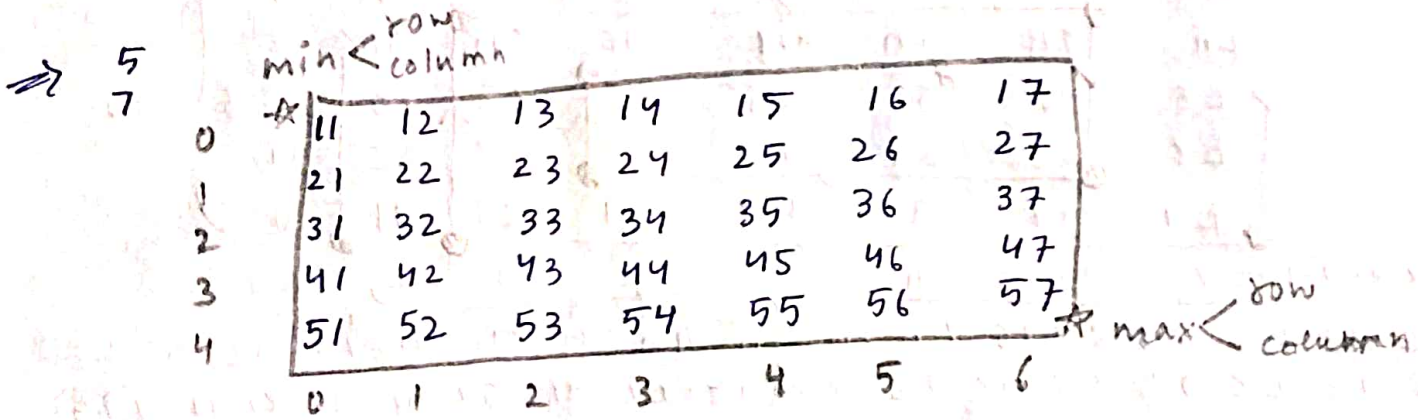
INPUT : 4 4 11 12 13 14 21 22 23 24 31 32 33 34 41 42 43 44

4 4 → n m → row column

numbers to be filled in 2d array



DABBE MEIN BABBA



1 EK BOX PURA KARNE KE LIYE, 4 WALL HONGI . 4 FOR LOOP LAGENGY.

- Left wall < column → Fixed → min ; row → varies → min se max
- bottom wall < row → Fixed → max ; column → varies → min se max
- Right wall < column → Fixed → max ; row → varies → max se min
- Top wall < row → fixed → min ; column → varies → max se min

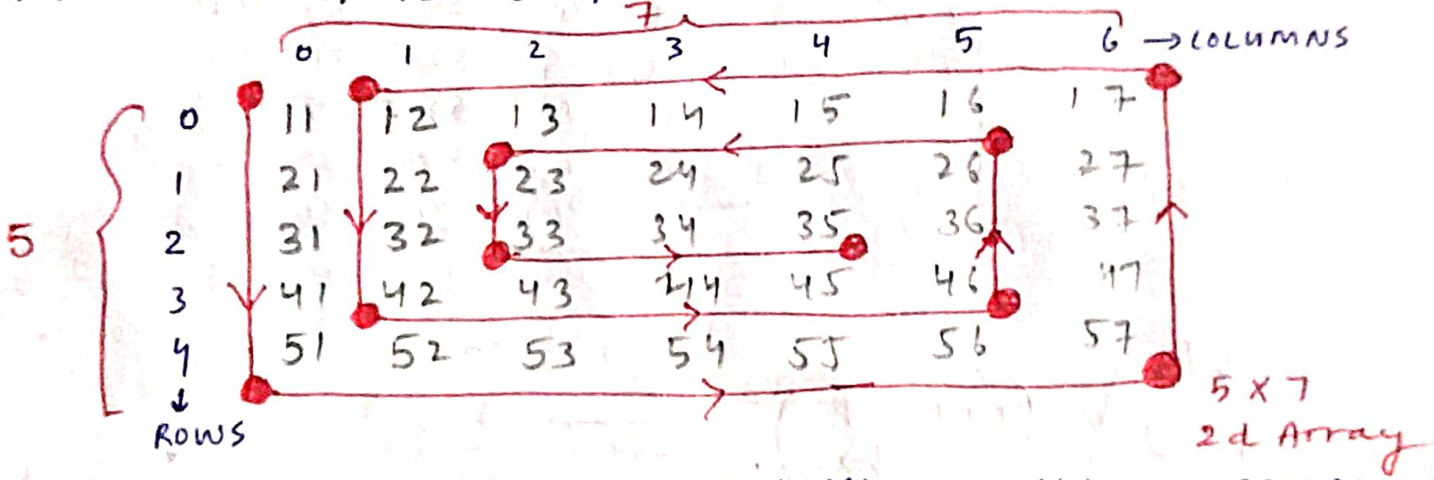
APPROACH

```
ps v m(s[] a){
  Scanner s = new Scanner(System.in);
  int n = s.nextInt();
  int m = s.nextInt();
  int [][] arr = new int[n][m];
  for(int i=0; i<arr.length; i++){
    for(int j=0; j<arr[0].length; j++){
      arr[i][j] = s.nextInt();
    }
  }
```

ARRAY BANAYA AUR FILL KRA !

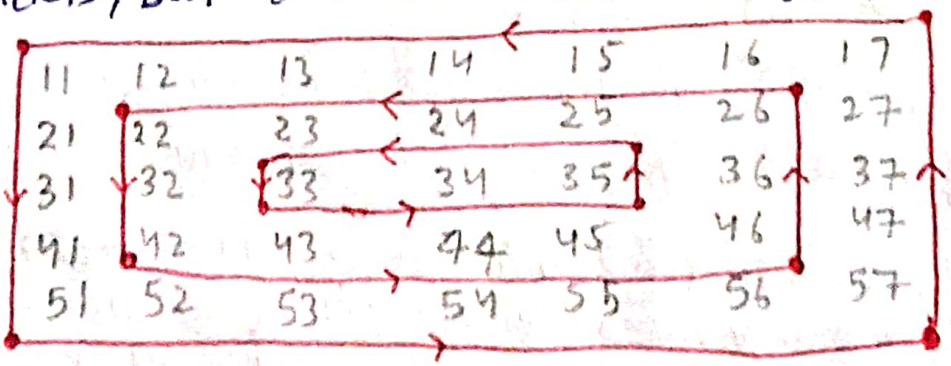
LET'S ASSUME INPUT IS :

5 7 11 12 13 14 15 16 17 21 22 23 24 25 26 27 31 32 33 34 35 36 37 41 42 43 44 45 46 47 51 52 53 54 55 56 57



TRICK : INSTEAD OF TRAVESING 2D-ARRAY ACTUALLY IN SPIRAL PATTERN,
WE MAY THINK OF TRAVESING IT BOX BY BOX MAKING SURE THAT VALUES AT CORNER DOESN'T REPEAT.

- STARTING WITH AN OUTER BOX, THEN MOVING TO THE SMALLER ONE. THIS WAY KEEP MOVING, UNLESS WE REACH THE LAST BOX. AND ITS NOT NECESSARY THAT THE LAST ONE IS A BOX, IT MAY BE JUST A COUPLE OF ELEMENTS, BUT CONSIDERED THEM AS BOXES AS SHOWN BELOW :



DABBE
ME
DABBA

```
int minr = 0; // mini. row
int minc = 0; // mini. column
int maxr = arr.length-1; // max. row
int maxc = arr[0].length-1; // max. column
```

IN 4 VARIABLES SE SUBSE OUTER MOST BOX DEFINE HOKA!

```
int tne = n*m; // total no. of elements
int count=0; // for keeping count of elements printed
```

min

column → Fix

row → vary → min to max

```
while (count < tne) {

[redacted]

for (int i=minr, j=minc; i<=minr && count<tne ; j++) { // LEFT WALL
    System.out.println (arr[i][j]);
    count++; // counter is incremented by 1
}
minc++; // updating minimum column
```

column → vary → min to max

row → Fix → max

```
// BOTTOM WALL
for (int i=maxr, j=minc; j<=maxc && count<tne ; j++) {
    System.out.println (arr[i][j]);
    count++; // counter is incremented by 1
}
maxr--; // updating maximum row
```

column → fix → max

row → vary → max to min

```
// RIGHT WALL
for (int i=maxr, j=maxc; i>=minr && count<tne ; i--) {
    System.out.println (arr[i][j]);
    count++; // counter is incremented by 1
}
maxc--; // updating maximum column
```

column → vary → max to min

row → Fixed → min

```
// TOP WALL
for (int i=minr, j=maxc; j>=minc [redacted] && count<tne ; j--) {
    System.out.println (arr[i][j]);
    count++; // counter is incremented by 1
}
minr++; // updating minimum row
}
}
```

```
if (count < tne) {
for(int i = minr; i<= maxr; i++){
    Syso (arr[i][minc]);
    count ++;
  }
```

SAME

```
for(int i=minr; i<= maxr &&
                    count<tne; i++)
{Syso (arr[i][minc]);
 count ++;
}
```

TIME COMPLEXITY

$O(n^2)$ → As there is nested ~~loop~~ for loop and the outer for loop runs (n) times.
There are two inner for loop.
Either (1) will run in each iteration.

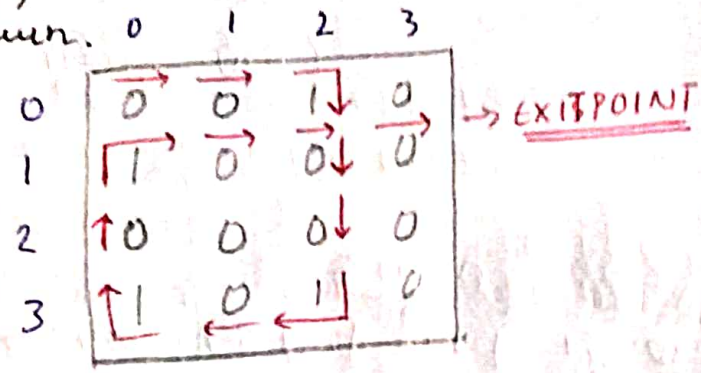
making the time complexicity : $O(n) * O(m) = O(n*m)$

SPACE COMPLEXITY

$O(1)$ → WE ARE not using any auxiliary space and hence the space complexity is $O(1)$.

EXIT POINT OF A MATRIX

The problem states that we have a matrix or a 2-d array that has only 1's and 0's as the values inside it. if we encounter a 0 in the matrix we have to keep moving in the direction that we are moving, otherwise, when we encounter 1, we will take a 90° right turn.



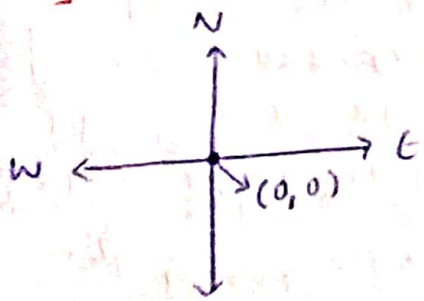
0 → pe sidha chalega

1 → pe 90° ka right turn lelega

APPROACH

```
P s v m (s [] a) {
  Scanner s = new Scanner (System.in);
  int r = s.nextInt();
  int c = ____________;
  int [][] arr = new int [r][c];
  for (int i=0; i<arr.length; i++) {
     arr[i][j] = s.nextInt();
  }
}
```

ARRAY
BANAYA
AUR
USSE
FILL
KRA



moving in E dirn —90° right turn→ South dirn
moving in S dirn —90° right turn→ West dirn
moving in W dirn —90° right turn→ North dirn
moving in N dirn —90° right turn→ East dirn

i = row
j = column

moving in East direction (→) j++
SOUTH (↓) i++
WEST (←) j--
NORTH (↑) i--

```
int i = 0;
int j = 0;
int dir = 0; // East = 0 ; South = 1 ; West = 2 ; North = 3
while (true) {
dir = (dir + arr[i][j]) % 4; // To keep the values b/w 0 and 3
if (dir == 0) { // EAST DIRECTION
    j++ ; // move in the same row
  if (j == arr[0].length) {
        j--;
        break;
    }
} else if (dir == 1){ // SOUTH DIRECTION
   i++; // move down the column
  if (i == arr.length){
        i--;
        break;
    }
} else if (dir == 2) { // WEST DIRECTION
   j--; // move backwards in the row
   if (j == -1) {
        j++;
        break;
    }
} else { // NORTH DIRECTION
    i--; // move up in the column.
    if (i == -1) {
      i++;
      break;
    }
}
}
```

→ Exit point is on column back (in last column)

→ Exit point is one row back (in last row)

→ Exit point is one column ahead (in 0th column)

→ Exit point is one row ahead (in 0th row)

TIME COMPLEXITY : $O(n^2)$

SPACE COMPLEXITY : $O(1)$

ROTATE BY 90 DEGREE

WE ARE GIVEN A SQUARE MATRIX AND WE HAVE TO ROTATE IT BY 90° DEGREE.

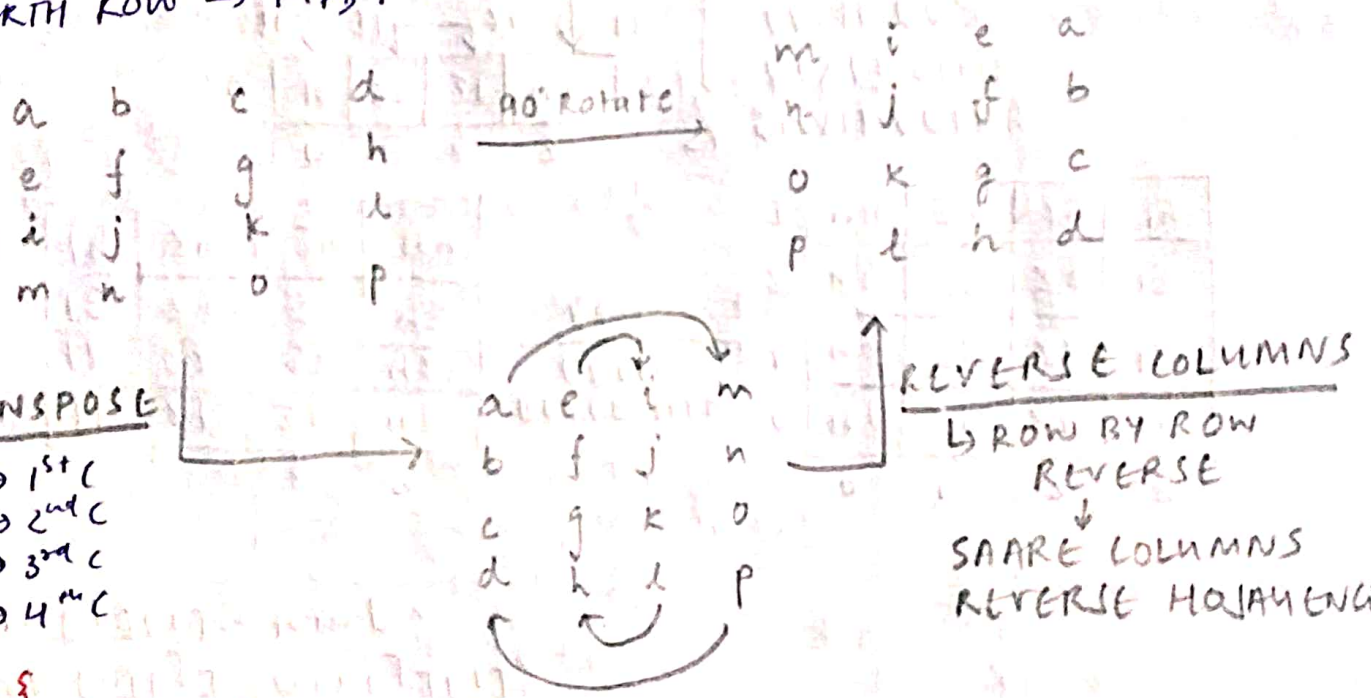
↳ n = m
↳ 1 input

	0	1	2	3
0	11	12	13	14
1	21	22	23	24
2	31	32	33	34
3	41	42	43	44

AFTER 90° ROTATION

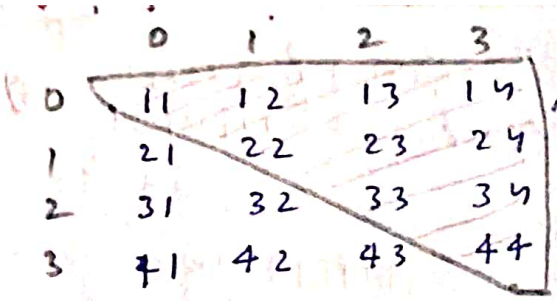
	0	1	2	3
0	41	31	21	11
1	42	32	22	12
2	43	33	23	13
3	44	34	24	14

// FIRST ROW ⟶ LAST COLUMN
// SECOND ROW → SECOND LAST COLUMN
// THIRD ROW → THIRD LAST COLUMN / SECOND COLUMN
// FOURTH ROW → FIRST COLUMN




```
p c m {
p s v m (s[]a){
  Scanner s = new Scanner(System.in);
  int n = s.nextInt();
  int [][]arr = new int[n][n];
  for(int i=0; i<arr.length; i++){
    for(int j=0; j<arr[0].length; j++){
      arr[i][j] = s.nextInt();
    }
  }
  rotateby90(arr);
  display(arr);
}
```

INPUT FROM USER AND 2D ARRAY CREATION



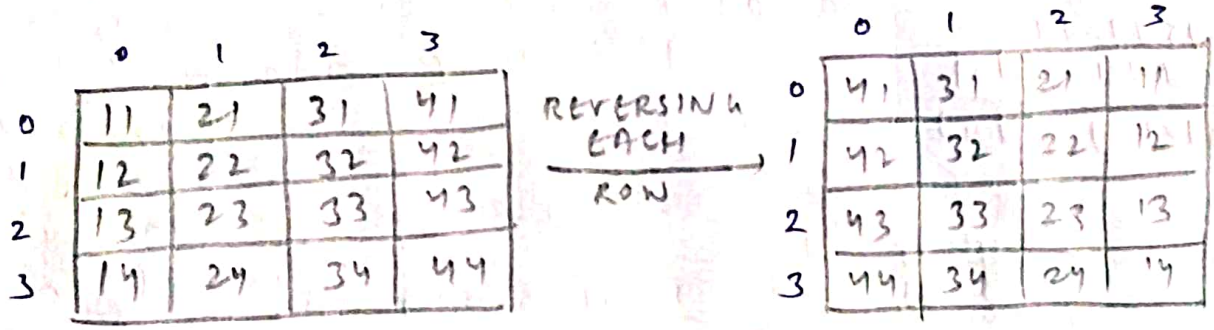
TRAVERSE ONLY THIS PART FOR TRANSPOSE. IT WILL PROVIDE THE RIGHT PATTERN. ★

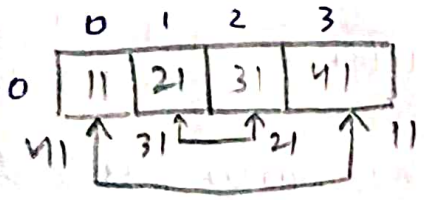
BUT IF WE TRAVERSE COMPLETE ARRAY THEN MANY PLACES WILL BE REMAINED UNCHANGED AND ANSWER WILL BE DIFFERENT.

```
PS  v rotateby90 (int [][] arr) {
for(int i=0 ; i < arr.length ; i++) {
   for(int j=0; j < i ; j++) {
    int temp = arr[i][j];
    arr[i][j] = arr[j][i];
    arr[j][i] = temp;
   }
}
```

arr[0].length

TRANSPOSE OF MATRIX





SIMILARLY REVERSING EACH ROW

```
for (int i=0; i < arr.length; i++){
 int li = 0;  //LEFT INDEX
 int ri = arr[0].length-1; //RIGHT INDEX
 while ( li <= ri) {
    int temp = arr[i][li];
    arr[i][li] = arr[i][ri];
    arr[i][ri] = temp;
    li++;
    ri--;
 }
}
}
```

REVERSE COLUMNS, ROW BY ROW ELEMENTS REVERSE

```
psv display(int[][] arr){
  for(int i=0; i<arr.length; i++){
    for(int j=0; j<arr[0].length; j++){
        System.out.print(arr[i][j] + " ");
    }
    System.out.println();
  }
}
}
```

DISPLAY FUNCTION TO DISPLAY THE FINAL MATRIX/ ARRAY

TIME COMPLEXITY

THE TIME COMPLEXITY FOR ROTATING THE MATRIX BY THIS PROCEDURE IS $O(n^2)$.

AS FOR TRANSPOSING WE ARE TRAVERSING THE UPPER TRIANGLE. i.e. WE ARE TRAVERSING $(n^2/2)$ ELEMENTS AND THEN REVERSING TAKES $O(n)$ time. AND WE HAVE TO REVERSE (n) ROWS.

SO IT BECOMES $O(n^2 + \frac{n^2}{2})$

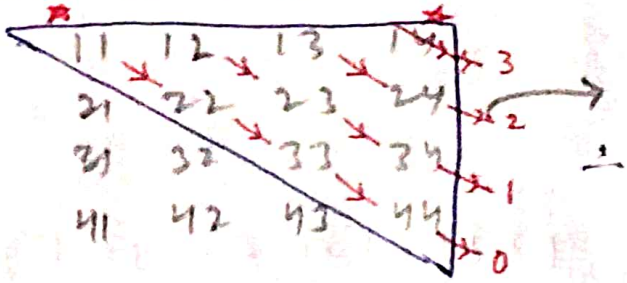
WHICH IS $O(n^2)$.

SPACE COMPLEXITY

THE SPACE COMPLEXITY IS $O(1)$ AS WE ARE TRANSPOSING AND THEN REVERSING IN-PLACE AND WE ARE NOT CONSUMING ANY AUXILIARY MEMORY.

THE STATE OF WAKANDA - 2 (DIAGONAL TRAVERSAL)

WE NEED TO PRINT UPPER DIAGONAL OF THE MATRIX



11, 22, 33, 44, 12, 23, 34, 13, 24, 14

∴ IT IS A SQUARE MATRIX

↳ n = m
row column

```
public static void main(String[] args){
    Scanner s = new Scanner(System.in);
    int n = s.nextInt();
    int[][] arr = new int[n][n];
    for(int i=0; i<arr.length; i++){
        for(int j=0; j<arr[0].length; j++){
            arr[i][j] = s.nextInt();
        }
    }
```

INPUT AND ARRAY CREATION

SABSE PHELE 0th DIAGONAL PRINT HOGA FIR 1ST, 2nd, 3rd.
HAR DIAGONAL KI SHURUWAT (i=0) HOTI HAI,
AUR (j = gap)
CONDITION → (j < n) HAR DIAGONAL KE LIYE!

```
for(int g=0; g<arr.length; g++){
    for(int i=0, j=g; j<arr.length; i++, j++){
        System.out.println(arr[i][j]);
    }
}
}
```

gap

loop for printing upper diagonal matrix

Time Complexity

WHEN
gap = 0, inner loop runs (n) times
gap = 1, (n-1) times
gap = 2, (n-2) times
.
.

n+(n-1)+(n-2)+(n-3)+... times

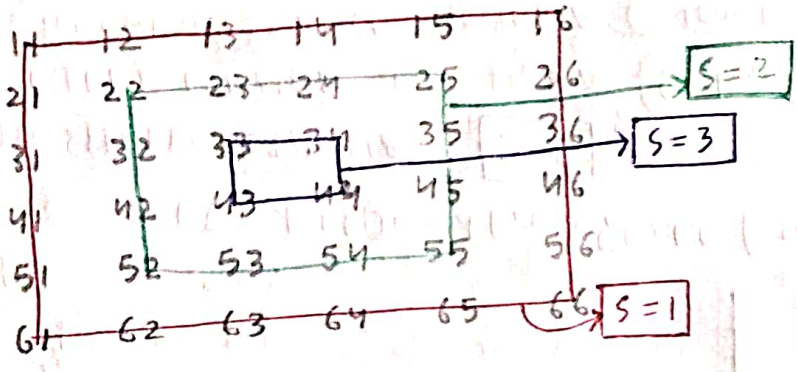
$= \frac{n*(n+1)}{2} \simeq O(n^2)$

SPACE COMPLEXITY

we are not using any auxiliary space and hence the space complexity O(1).

⑨ RING ROTATE OR SHELL ROTATE

WE HAVE A [n x m] MATRIX WHERE (n) IS THE NUMBER OF ROWS AND (m) IS THE NUMBER OF COLUMNS. WE WILL BE GIVEN A RING NUMBER (s) REPRESENTING THE RING OF THE MATRIX.
WE WILL BE GIVEN A NUMBER (r) REPRESENTING THE NUMBER OF ROTATIONS IN AN ANTI-CLOCKWISE MANNER OF THE SPECIFIED RING.
WE ARE REQUIRED TO ROTATE THE (s^{th}) RING BY (r) ROTATIONS AND DISPLAY THE ROTATED MATRIX



SHELL KE DATA KO 1-D ARRAY ME BHAR LENGY, [SHELL convert → 1-D]
1-D ARRAY ROTATE KRENGY, (ROTATE 1-D Array)
ROTATED 1-D ARRAY KO WAPIS SHELL KO BHAR DENGY.
[ROTATED 1-D ARRAY convert → SHELL]

⑩ PSEUDOCODE

```
oned = fillOnedFromShell(arr, s);
rotate(oned, r);
fillShellFromOned(arr, s, oned);

p c m {
p s v m(s[]a){
  Scanner s = new Scanner(System.in);
    int n = s.nextInt();
    int m = s.nextInt();
    int[][] arr = new int[n][m];
    for(int i=0; i<arr.length; i++){
        for(int j=0; j<arr[0].length; j++){
            arr[i][j] = s.nextInt();
        }
    }
  }
  int s = s.nextInt(); // SHELL
  int r = s.nextInt(); // ROTATION
  rotateShell(arr, s, r);
  display(arr);
}
```

INPUT FROM USER AND ARRAY CREATION

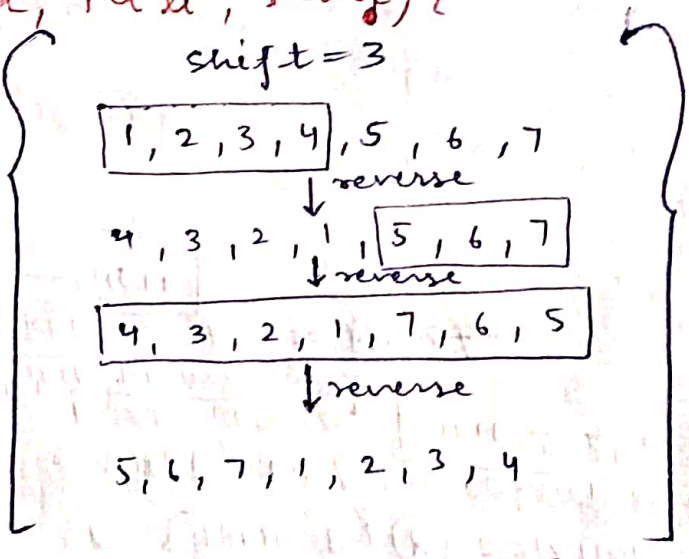
```
p s v rotateShell (int[][] arr, int s, int r) { // To rotate
    int[] oned = fillOnedfromShell (arr, s);      the particular
    rotate (oned, r);                              shell
    fillShellfromOned (arr, s, oned);
}
```

```
p s v  rotate (int[] oned, int r) { // rotate function
    r = r % arr.length;
    if (r < 0) {
        r = r + arr.length;
    }
    reverse (oned, 0, oned.length - r - 1);
    reverse (oned, oned.length - r, oned.length - 1);
    reverse (oned, 0, oned.length - 1);
}
```

↓
To rotate the ~~function~~ oned array by (r) given by user

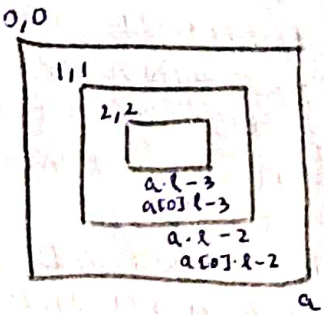
```
p s v  reverse (int [] oned, int li, int ri) {
    while (li < ri) {
        int temp = oned[li];
        oned[li] = oned[ri];
        oned[ri] = temp;
        li++;
        ri--;
    }
}
```

// Reverse function → To reverse the oned.




```
p s  int[] fillOnedfromShell (int[][] arr, int s) {
```

JO SHELL HOTA HAI [S=3],
TOP LEFT CORNER WO HOGA (S-1, S-1)
BOTTOM RIGHT CORNER WO HOGA (a.length-S, a[0].l-S)
YEH HAR JAGH HOGA
HAR SHELL KE LIYE,



```
int minr = s-1;
int minc = s-1;
int maxr = arr.length - s;
int maxc = arr[0].length - s;
// int sz = lw + bw + rw + tw - 4;
//int sz = 2*(maxr-minr+1)+2*(maxc-minc+1) -4;
int sz = 2*(maxr-minr)+2*(maxc-minc);
int [] oned = new int[sz];
int idx = 0;
for(int i=minr, j=minc; i<=maxr; i++){
    oned[idx] = arr[i][j];
    idx++;
}
for(int i=maxr, j=minc+1; j<=maxc; j++){
    oned[idx] = arr[i][j];
    idx++;
}
for(int i=maxr-1, j=maxc; i>=minr; i--){
    oned[idx] = arr[i][j];
    idx++;
}
for(int i=minr, j=maxc-1; j>=minc+1; j--){
    oned[idx] = arr[i][j];
    idx++;
}
return oned;
}
```

same same

ONE-D ARRAY KO SIZE KITNA HOGA ?

↳ SHELL KE SAARE ELEMENT HONGY PAR CORNERS DO BAAR LENA NAHI !

total element

= high - low + 1
= 5 - 1 + 1
= 4 + 1
= 5

Left wall
Row → increase → minr → maxr
Col. → stay at minc

bottom wall
Row → stays at maxr
Col. → ↑ → minc to maxc

Right wall
Row → ↓ → maxr to minr
Col. → stay at maxc

top wall
rows stay at minr
Col. → ↓ → maxc to minc

```
p s v fillShellFromOned (int[][] arr, int s, int[] oned) {
    int minr = s - 1;
    int minc = s - 1;
    int maxr = arr.length - s;
    int maxc = arr[0].length - s;
    int idx = 0;
    for(int i = minr, j = minc; i <= maxr; i++) {
        arr[i][j] = oned[idx];
        idx++;
    }
    for(int i = maxr, j = minc + 1; j <= maxc; j++) {
        arr[i][j] = oned[idx];
        idx++;
    }
    for(int i = maxr - 1, j = maxc; i >= minr; i--) {
        arr[i][j] = oned[idx];
        idx++;
    }
    for(int i = minr, j = maxc - 1; j > minc; j--) {
        arr[i][j] = oned[idx];
        idx++;
    }
}
```

// Oned array se shell fill kiya !

// left wall

// bottom wall

// Right wall

// Top wall

```
p s v display (int[][] arr) {
    for(int i = 0; i < arr.length; i++) {
        for(int j = 0; j < arr[0].length; j++) {
            System.out.print(arr[i][j] + " ");
        }
        System.out.println();
    }
}
```

// display function

↳ To display the Ring rotated array !

Time complexity → O(n + m)
Space complexity → O(n+m)

SADDLE PRICE → (POINT)

// YOU ARE GIVEN A SQUARE MATRIX OF SIZE (n). YOU ARE GIVEN (n*n) ELEMENTS OF THE SQUARE MATRIX.

// YOU ARE REQUIRED TO FIND THE SADDLE PRICE OF THE GIVEN MATRIX AND PRINT THE SADDLE PRICE.

// THE SADDLE PRICE IS DEFINED AS THE LEAST PRICE IN THE ROW BUT THE MAXIMUM PRICE IN THE COLUMN OF THE MATRIX.

✓ 2 SADDLE POINTS NAHI HO SAKTY!

✓ SADDLE POINT → ESSA NO. JO APNI ROW ME MIN. (CHOTA) HO! AUR APNE COLUMN ME MAX. (BADA) HO!

a	b	c	d
e	f*	g	h
i	j	k	l*
m	n	o	p

Let us say (f) and (l) are Saddle points

✓ Agar (f) saddle point hai toh apni row ke least hai toh vo certainly (h) se chota hoga! Aur apne column ka max hai toh (j) se bada hoga!

$$j < f < h \quad \text{—(1)}$$

↳ This like states that

$\{j < h\}$

$\{h < j\}$] → This not possible at all.

✓ Aur (l) bhi saddle point hai! toh vo apni row me least hai toh (j) se chota hoga! column me max hai toh (h) se bada hoga!

$$h < l < j \quad \text{—(2)}$$ → This like states that

11	12	13	14
21	22	23	24
31	32	33	34
41*	42	43	44

✓ Esse har row ko travel krenge aur check krenge!

SADDLE POINT → Apni row ke least
Apni column ke max

✓ Phele puri row ko travel krengy usme uska least dhunde gengy! toh hume potential saddle point milega!

✓ hum potential saddle point ko check krengy kya to apne column me max hai: toh to 2 Potentially se Saddle point hojeyega!

Q ESSA ho sakta hai ki ek bhi saddle point na mile! → YES
Is Point invalid input

```
p c  m {
p s  v  m (S [] a) + t {
Scanner s= new Scanner (System.in);
 int n = s.nextInt();
 int [] [] arr = new int [n] [n];
 for (int i =0; i<arr.length; i++) {
     for (int j=0 ; j<arr[0].length ; j++) {
         arr[i][j] = S.nextInt();
     }
 }
// Saari rows me loop lagya!
 for (int i = 0; i<arr.length ; i++) {
   // mujhe per row ka mini chaiye!
   // fir usse check krna hai ki column max. hai ki nahi!
   int lci = 0;                          → smallest value (j) (svj)
   for (int j = 1; j<arr[0].length; j++) {
     if ( arr[i][j] < arr[i][lci]) {
           lci = j;
     }
   }
 }
```

Code written with green color is correct code with all test cases pass.

Time Complexity
In worst case, we might need to travel all the elements in the matrix. - O(n²)

Space Complexity : O(1)

int min = arr[i][0];
min
min = arr[i][j];

```
   boolean flag = true;
// Ab mujhe usse column check krna hai ki yeh max hai ya nahi? Dubara loop lagana hai
for (int k= 0; k< arr.length; k++) {
    if (arr[k][lci] > arr[i][lci]) {
          flag = false;
          break;
    }
}
if (flag == true) {
    Syso (arr[i][lci]);
    return;
}
} Syso ("INVALID INPUT");
}
```

→ hum saddle point kudh ko manke chalte hai!

min

// Agar flage true raha toh array print krenge

// Saddle point

// warna yeh print hoga!

SEARCH IN A SORTED 2-D ARRAY

HUME EK ARRAY MILA HAI! Sorted hai (row + Columns).

11	12	13	14 ↓ x
21	22	23	24 ↓
31	32 ←	33 ←	34
41	42	43	44

∴ x → chota hai toh left me jaye!
→ bada hai toh niche (down) jaye!
→ same hai toh milgaya (FOUND!)!

Let find (32)
Binary Search jaisa kuch lagyengy!

42 → FIND → ?
40 → INVALID

// Hume Square matrix mila hai!

x = ?

```
p cm {
p s v m [s[] a) + e{
Scanner s = new scanner (System.in);
  int n = s.nextInt();
  int [][] arr = new int [n][n];
for (int i=0; i<arr.length; i++) {
  for (int j=0; j<arr[0].length; j++) {
        arr[i][j] = s.nextInt();
  }
}
int (x) = s.nextInt();
```

Input le liya!

x → data → Jo search krna hai usska input

// Aab hume chej dundhni hai!

```
int (i) = 0;
int (j) = arr[0].length - 1;
```

row → i, column → j] Corner milgaya!

m → Right-top-most column.

```
while (i < (arr.length) && j >= 0) {
  if ((x) == arr[i][j]) {
      System.out.println (i);
      (j); return;
  } else if (x < arr[i][j]) {
      j--;
  } else {
      i++;
  }
} System ("NOT FOUND");
}
```

Approach

To be Found

Data = x = 12

- Can enter from 4 corners
- move along the rows and columns not diagonals.

∴ Let say we have to Find Data = 15 ☆

TO BE FOUND

2 (00)	3 (01)	5 (02)	8 (03)
6 (10)	9 (11)	12 (12)	18 (13)
11 (20)	15 (21)	20 (22)	25 (23)
23 (30)	30 (31)	40 (32)	50 (33)

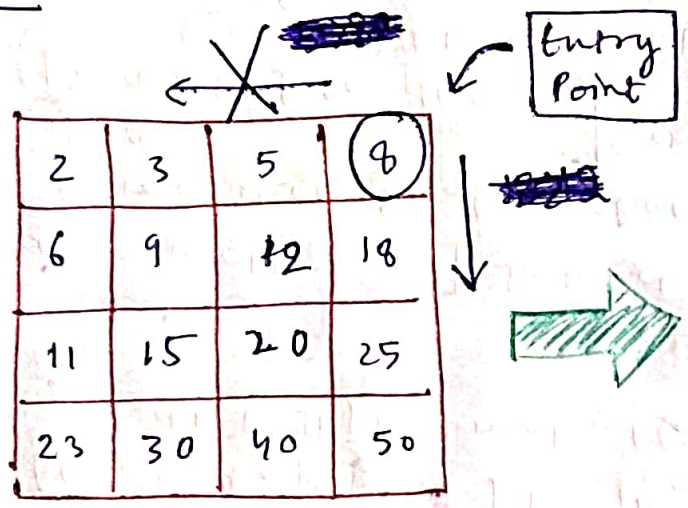
3 < 12 →

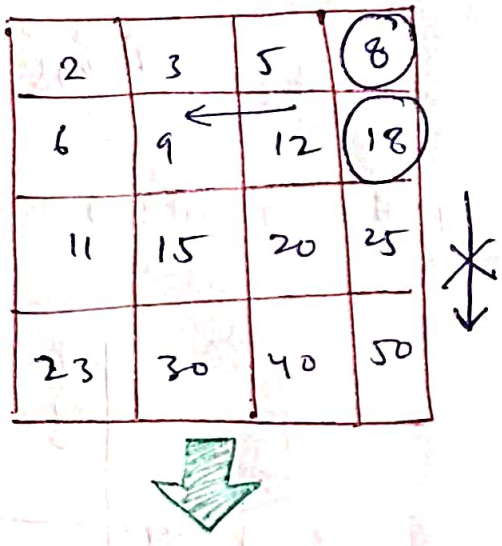
6 < 12 ↓

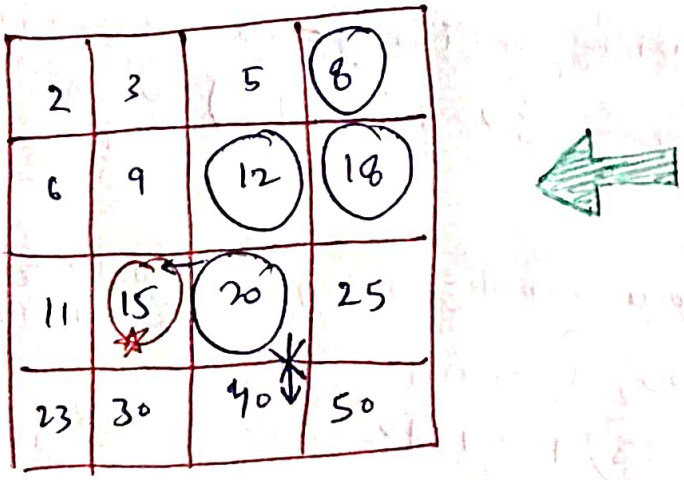
25 > 12 ↑

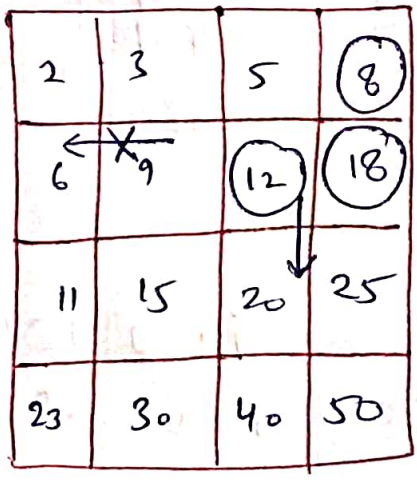
40 > 12 ←

Entry Point Decision









Time Complexity: Drops from O(m*n) to O(m+n)

Space Complexity: O(1)