MATRICES

$\begin{bmatrix} 7 \\ 2 \end{bmatrix}$ ⟶ [[7], [2]]

2×1 matrix

2D array.

$\begin{bmatrix} 0 & 1 & 3 \\ 9 & 5 & 8 \end{bmatrix}$ ⟶ [[0, 1, 3], [9, 5, 8]]

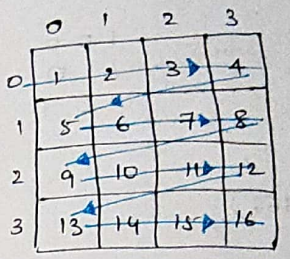
Matrix transformations :-

Operations performed on matrices that result in a new matrix

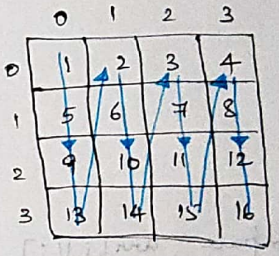
1. Addition & Subtraction.
2. Multiplication.
3. Inverse. $A \cdot A^{-1} = I$
4. Transpose. $\begin{bmatrix} 0 & 1 & 3 \\ 9 & 5 & 8 \end{bmatrix}_{2\times3} \rightarrow \begin{bmatrix} 0 & 9 \\ 1 & 5 \\ 3 & 8 \end{bmatrix}_{3\times2}$
5. Scalar multiplication. $\begin{bmatrix} 0 & 1 & 3 \\ 9 & 5 & 8 \end{bmatrix} \times 3 = \begin{bmatrix} 0 & 3 & 9 \\ 27 & 15 & 24 \end{bmatrix}$
6. Rotation. $\begin{bmatrix} 0 & 1 & 3 \\ 9 & 5 & 8 \\ 9 & 5 & 8 \end{bmatrix}$ —90° clockwise→ $\begin{bmatrix} 9 & 9 & 0 \\ 5 & 5 & 1 \\ 8 & 8 & 3 \end{bmatrix}$
7. Reflection. $\begin{bmatrix} 0 & 1 & 3 \\ 9 & 5 & 8 \\ 9 & 5 & 8 \end{bmatrix}$ —x-axis→ $\begin{bmatrix} 9 & 5 & 8 \\ 9 & 5 & 8 \\ 0 & 1 & 3 \end{bmatrix}$ —y-axis→ $\begin{bmatrix} 3 & 1 & 0 \\ 8 & 5 & 9 \\ 8 & 5 & 9 \end{bmatrix}$

Matrix traversal :-

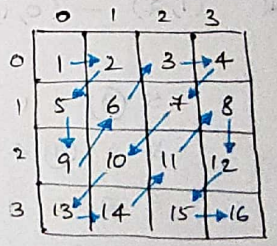
① Row-major traversal :-



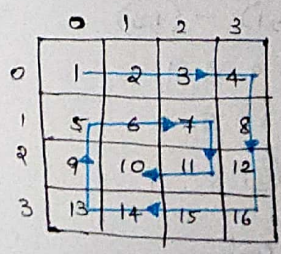
② Column-major traversal :-



③ Diagonal traversal :-



④ Spiral traversal :-



Examples :-

① Rotate and invert an image :- (LC48 for rotate by 90°)

```
1 1 0        0 1 1        1 0 0
1 0 1  90°   0 0 1  invert 1 1 0
0 0 0  ----> 0 1 0  ------> 1 0 1
       clockwise
```

② Toeplitz matrix :- (LC766)

Toeplitz → every decreasing diagonal from left to right (or every left diagonal) has the same elements.

```
5 9 4 1
2 5 9 4
3 2 5 9
```

Row major traversal
(i,j) Starting from (1,1) compare arr[i][j] with arr[i-1][j-1].

* (1,1) — (0,0)

```
(5) 9 4 1
2 (5) 9 4
3 2 5 9
```

continue.

* (1,2) — (0,1)

```
5 (9) 4 1
2 5 (9) 4
3 2 5 9
```

continue.

* (1,3) — (0,2)

```
5 9 (4) 1
2 5 9 (4)
3 2 5 9
```

continue

* (2,1) — (1,0)

```
5 9 4 1
(2) 5 9 4
3 (2) 5 9
```

continue

* (2,2) — (1,1)

```
5 9 4 1
2 (5) 9 4
3 2 (5) 9
```

continue

* (2,3) — (1,2)

```
5 9 4 1
2 5 (9) 4
3 2 5 (9)
```

Because we have successfully traversed the entire matrix, we return TRUE, since all left diagonals contain identical elements.

Problem: Sparse Matrix Multiplication

(i) * Sparse Matrix :-

Sparse matrices are matrices in which most elements are zero. To save space and running time, it is critical to only store the non-zero elements.

A standard representation of sparse matrix is to use an array with one element per row, each of which contains a linked-list of the non-zero values, in that row along with their column number.

Sparse matrix: $A = \begin{bmatrix} 2.0 & -1.0 & 0 & 0 \\ -1.0 & 2.0 & -1.0 & 0 \\ 0 & -1.0 & 2.0 & -1.0 \\ 0 & 0 & -1.0 & 2.0 \end{bmatrix}$

Representation (Row Map):

[{(0, 2.0), (1, -1.0)},
{(0, -1.0), (1, 2.0), (2, -1.0)},
{(1, -1.0), (2, 2.0), (3, -1.0)},
{(2, -1.0), (3, 2.0)}]

~~List<Pair> rowMapToValPairs =~~

List<Pair>[] rowMap = new ArrayList[r] ;

r = no. of rows, Pair => { colIndex, value }

Similarly,

List<Pair>[] colMap = new ArrayList[c] ;

c = no. of columns, Pair => { rowIndex, value }

* Matrix Multiplication :-

mat 1 $r_1 \times m_1$ mat 2 $r_2 \times m_2$

matrix multiplication is possible only if $m_1 = r_2$.

mat 1 $r_1 \times m_1$ × mat 2 $r_2 \times m_2$ = mat $r_1 \times m_2$.

eg:-

A

r_1	2	0	1
r_2	10	20	2

2 × 3

×

B

c_1	c_2	c_3	c_4
5	1	1	10
1	1	5	1
10	5	1	1

3 × 4

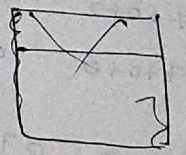
= ? (ans)

Approach:

ans =

$r_1 \times c_1$	$r_1 \times c_2$	$r_1 \times c_3$	$r_1 \times c_4$
$r_2 \times c_1$	$r_2 \times c_2$	$r_3 \times c_3$	$r_4 \times c_4$

2 × 4



$r_1 \cdot c_1 = (2 \times 5) + (0 \times 1) + (1 \times 10)$ → dot product.

$= 10 + 0 + 10$

$= 20$

$r_2 \cdot c_2 = (2 \times 1) + (0 \times 1) + (1 \times 5)$

$= 2 + 0 + 5$

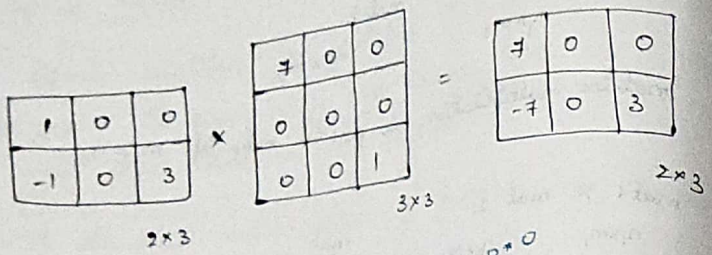
$= 7$

⋮

$\Rightarrow ans_{00} = A_{00} \times B_{00} + A_{01} \times B_{10} + A_{02} \times B_{20}$

$ans_{ij} = \sum_{k=0}^{n-1} A_{ik} B_{kj}$

* Sparse Matrix Multiplication :-



Matrix multiplication :- dot product ⇒ $1*7+0*0+0*0 = 7+0+0 = 7$

$$\begin{bmatrix} (1,0,0)\cdot\begin{pmatrix}7\\0\\0\end{pmatrix} & (1,0,0)\begin{pmatrix}0\\0\\0\end{pmatrix} & (1,0,0)\begin{pmatrix}0\\0\\1\end{pmatrix} \\ (-1\ 0\ 3)\begin{pmatrix}7\\0\\0\end{pmatrix} & (-1\ 0\ 3)\begin{pmatrix}0\\0\\0\end{pmatrix} & (-1\ 0\ 3)\begin{pmatrix}0\\0\\1\end{pmatrix} \end{bmatrix}$$

$$= \begin{bmatrix} 7+0+0 & 0+0+0 & 0+0+0 \\ -7+0+0 & 0+0+0 & 0+0+3 \end{bmatrix}$$

$$= \begin{bmatrix} 7 & 0 & 0 \\ -7 & 0 & 3 \end{bmatrix}$$

As you can see, in the above multiplication, there are many unuseful multiplications involving zeroes.

So we use compression technique.

We compress matrix A to row map

matrix B to col map

$$A = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 0 & 3 \end{bmatrix} \qquad B = \begin{bmatrix} 7 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

rowMap[] = [{(0,1)}, {(0,-1),(2,3)}]

colMap = [{(0,7)}, { }, {(2,1)}]

size of rowMap[] = 2 ⇒ 2 rows.

size of colMap[] = 3 ⇒ 3 columns.

we multiply each "row" of rowMap to each "col" of colMap and the dot product condition is

when colIndex = rowIndex.

$r_1 \times c_1$

$$\begin{bmatrix} (0,1)*(0,7) & - & - \\ r_2 \times c_1 & & r_3 \times c_3 \\ (0,-1)*(0,7) & - & (2,3)*(2,1) \end{bmatrix}$$

$$= \begin{bmatrix} 1*7 & 0 & 0 \\ -1*7 & 0 & 3*1 \end{bmatrix}$$

$$= \begin{bmatrix} 7 & 0 & 0 \\ -7 & 0 & 3 \end{bmatrix}$$

Note :-

colIndex is the first value in the pair of rowMap. e.g., in (0,1) → colIndex = 0

(0,-1) → colIndex = 0

(2,3) → colIndex = 2.

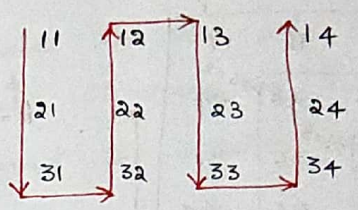
Similarly rowIndex is the first value in pair of colMap

e.g., (0,7) → rowIndex = 0 (2,1) → rowIndex = 2

By representing as rowMap & colMap, we eliminate multiplications with zeroes and thus reduce time complexity.

(ref: -11_LC311_SparseMatrixMultiplication.java)

Problem: Wave Traversal

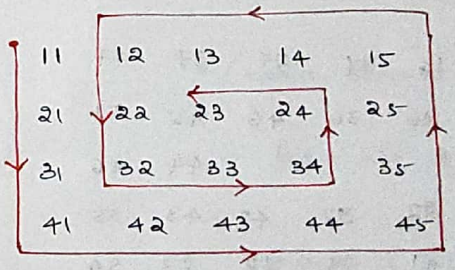


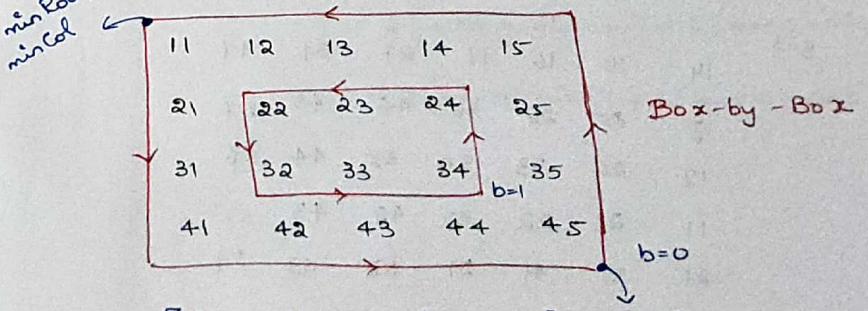
We print each column (0 to n-1) in different directions :-

↳ Even columns → top-to-bottom.

↳ Odd columns → bottom-to-top.

Problem: Spiral display.

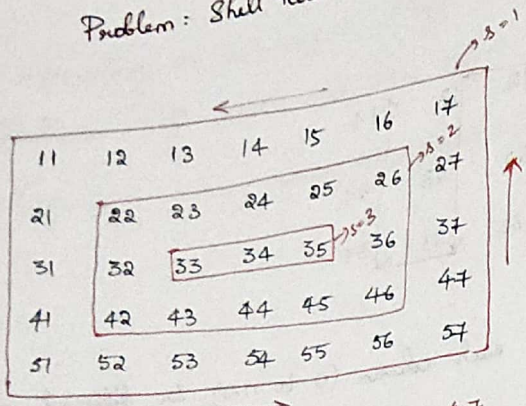




boundary 'o'

- left wall : r: minRow → maxRow
- bottom wall: c: minCol → maxCol
- right wall: r: maxRow → minRow
- top wall : c: maxCol → minCol.

Problem: Shell Rotate.



s = 1
k = 3

(1)

s=1 14 15 16 17 27 37 47
57
13
56
12
11 55
21 31 41 51 52 53 54

s=2 14 15 16 17 27 37 47
13 25 26 36 46 45 57
12 24 44 56
11 23 22 32 42 43 55
21 31 41 51 52 53 54

s=3 14 15 16 17 27 37 47
13 25 26 36 46 45 57
12 24 33 34 35 44 56
11 23 22 32 42 43 55
21 31 41 51 52 53 54

s=1
minrow = 0
1 2 3 4
5 6 7 8
maxrow = 2
9 10 11 12

leftwall size = (maxrow - minrow + 1)

Approach :-

(1) Convert the shell into 1D array.
(2) Rotate 1D array by k (anti-clockwise)
(3) Convert the rotated 1D array back to the shell.

For shell 's', top-left is $(s-1, s-1)$ (minr, minc)
bottom-right is $(n-s, m-s)$ (maxr, maxc)

size of shell, sz = left wall + bottom wall + right wall + top wall − 4

each wall repeats corner elements.

$= (\text{maxr} - \text{minr} + 1) + bw + rw + tw - 4$

$\because lw = rw$ & $bw = tw$.

$= 2 * (lw) + 2(bw) - 4$

$= 2(\text{maxr} - \text{minr} + 1) + 2(\text{maxc} - \text{minc} + 1) - 4$

$= 2(\text{maxr} - \text{minr}) + 2(\text{maxc} - \text{minc})$

Saddle point in a matrix

Find saddle point in a matrix.

A saddle point is an element of the matrix such that it is a minimum element in its row and maximum in its column.

Approach:

eg: arr[3][3] = $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$

7 is minimum in its row,
maximum in its col.

APPROACH:- (Optimized)

Proof that there is only one saddle point:

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

f → saddle => f < h & f > j

so j < f < h ——①

l → saddle => l < j & l > h

so h < l < j ——②

From ①, j < h

From ②, j > h

which is wrong.

Hence, more than one saddle point doesn't exist

Algorithm:-

① Find min element of current row and store col index of minimum element.

② Check if row min element is also maximum in its column. We use the stored col index here.

③ If yes, then saddle point else continues till the end.

Program:-

TC: O(mn)
SC: O(1)

```
public List<Integer> luckyNumbers (int[][] matrix) {
    int ansCol = 0;
    int ans = matrix[0][0];
    for (int i=0; i < matrix.length; i++) {
        int minElem = matrix[i][0];
        int colNo = 0;
        for (int j = 1; j < matrix[0].length; j++) {
            if (matrix[i][j] < minElem) {
                colNo = j;
                minElem = matrix[i][j];
            }
        }
        boolean isSolution = true;
        for (int k=0; k < matrix.length; k++) {
            if (minElem < matrix[k][colNo]) {
                isSolution = false;
                break;
            }
        }
```

```
        if (isSolution) {
            return List.of(minElem);
        }
    }
}
return Collections.emptyList();
}
```

Approach 2: Using Maximums & Minimums (Brute force)

TC: $O(mn)$

SC: $O(m+n)$

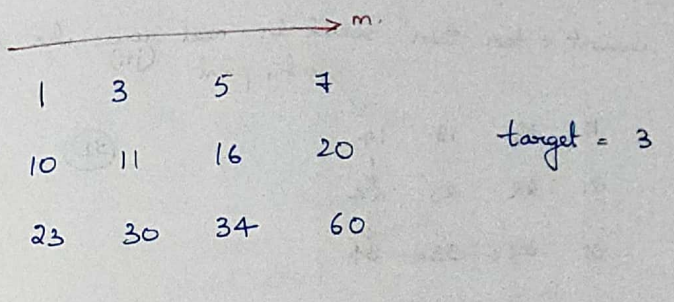
① Initialize row_minimum & col_maximum arrays with MAX_VALUE & MIN_VALUE.

② Loop over matrix elements to find & update row_minimum & col_maximum

③ Loop through matrix again to identify elements which is both row_minimum & col_maximum.

~~Approach 3: Find the only number with math.~~

Problem: Search in a 2D Sorted Array



Approach 1:- Find row first by using binary search on first column, then binary search on elements of row.

① if (tar > arr[mid][0])
if (tar < arr[mid+1][0] (&) mid == last row)

then search in the row no. "mid".
otherwise set left pointer to mid+1.

② if (tar < arr[mid][0]
& tar > arr[mid-1][0])

then search in row no. = "mid-1".

else if (mid == 0) then search in row no. = mid.

otherwise set right pointer to mid-1.

Time complexity

$O(\log n + \log m)$.

$= O(\log nm)$

Approach 2:- entry point element at (0, m-1)

if element < tar then search in next row, (j+1), else search in same row (i-1).

entry point

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

(32)

T.C : O(n+m).

Approach 3: Treat 2D matrix as 1D array.

	0	1	2	3	4
0	11	12	13	14	15
1	21	22	23	24	25
2	31	32	33	34	35
3	41	42	43	44	45

00	01	02	03	04	10	11	12	13	14
0	1	2	3	4	5	6	7	8	9
11	12	13	14	15	21	22	23	24	25
15	16	17	18	19	31	32	33	34	35
41	42	43	44	45					

n m	
00	0
04	4
10	5
11	6
12	7
20	10

We go to the beginning of next row, when current row 'm' elements are traversed. (x * m)

In current row, an element is found at column y.

(x * m) + y

	0	1	2	3
0	11	12	13	14
1	21	22	23	24
2	31	32	33	34.

(21)

2 rows already placed in 1D array.

2 * m

= 2 * 4

= 8

current row position.

⇒ 1 element of current row already placed in 1D array.

= (+1)

= 8 + 1

= 9.

∴ matrix[x][y] = arr[x * m + y]

32 should be placed at pos 9 of 1D array.

0	1	2	3	4	5	6	7	8	9	10	11
11	12	13	14	21	22	23	24	31	32	33	34

Similarly 1D array is placed into matrix as:

Say, current position, x = 9.

Let m is no. of columns ⇒ no. of elements in each row.

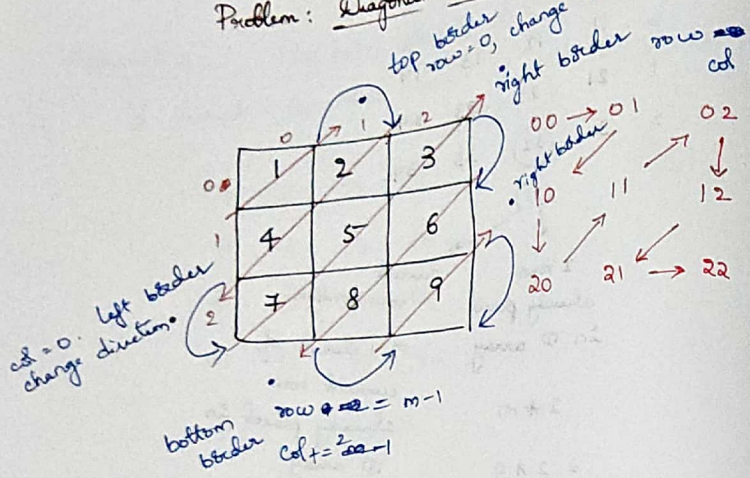
Hence we can determine which row arr[x] or arr[9] belongs to using

$$\text{row} = \frac{x}{m} = \frac{9}{4} = 2.$$

and the remaining positions (x % m) = 9 % 4 = 1 gives the position of element in that row (arr[x])

$$\therefore \quad arr[x] = matrix\left[\frac{x}{m}\right][x \% m];$$

Problem: Diagonal Traversal



Approach 1:

We have four borders
- → top border
- → right border
- → bottom border
- → left border.

Now, there are few corners, say top-right. In such corner, right border takes pre-dominance as (setting row = 0, change direction) is not going to be correct. Instead follow right border rule

row += 2 &

col = n-1.

Which means that we need to predominantly check right border before top border.

Similarly bottom-left We need to predominantly check bottom border before left border.

right | bottom
top. | left

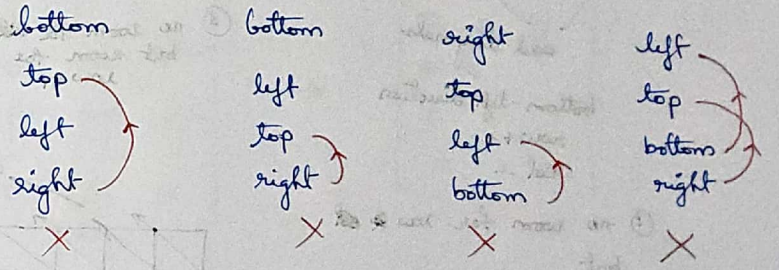
Hence the order should be maintained as follows.

"right before top border"

"bottom before left border"

eg:-

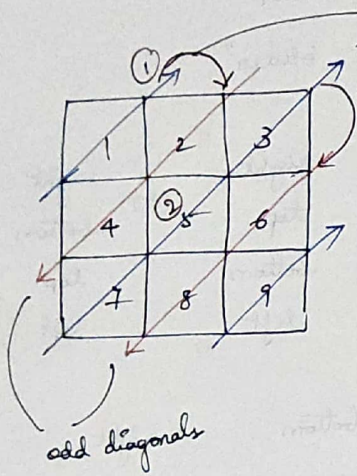
bottom (or) right (or) right (or) bottom
left top bottom right
right bottom top top
top ✓ left ✓ left ✓ left ✓



Note:-

top-left & bottom-right directions are not applicable. Since traversal is only ↗↙ directions

Approach 2:-



even diagonals

top-right direction. row-- col++

① no room for row but room for col
col++

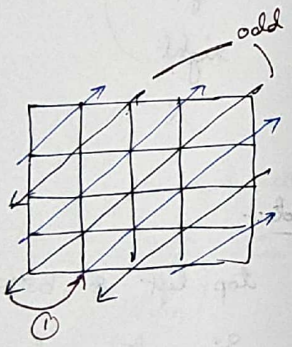
② room for both row & col
row --
col ++

③ no room for col but room for row.
row ++

odd diagonals

bottom-left direction
row ++
col --

① no room for row & col but

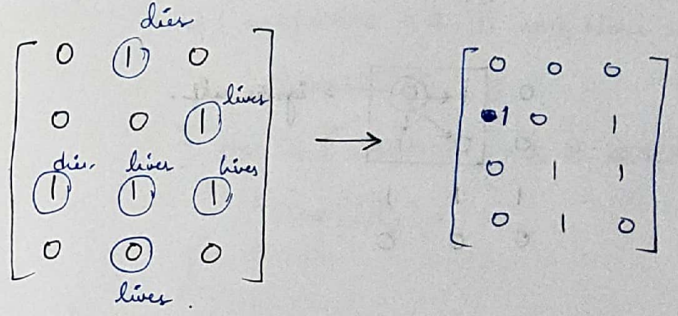


Problem: GAME OF LIFE (LC 289)

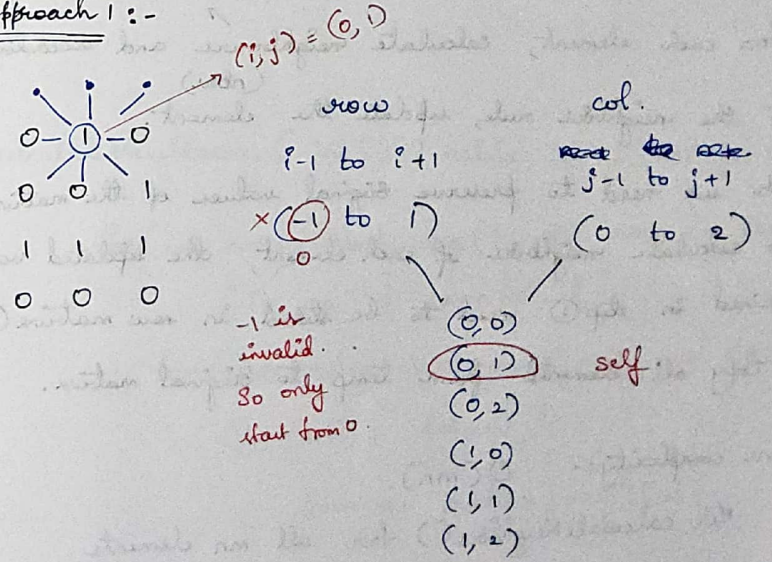
m×n grid
a cell: 0 → dead 1 → alive.

→ < 2 nbrs dies (0) under population
→ = 2 & 3 nbrs lives (1) lives onto next generation.
→ > 3 nbrs dies (0) over population
→ = 3 nbrs becomes live cell (1) reproduction.

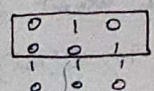
8 neighbors horizontal, vertical, diagonal



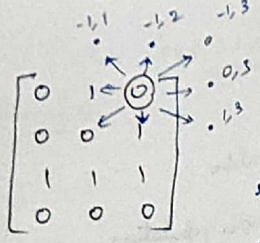
Approach 1:-



In the above example, for element at (0,1) we check from
row: 0 → 1 col: 0 → 2 ⇒



but we subtract self.

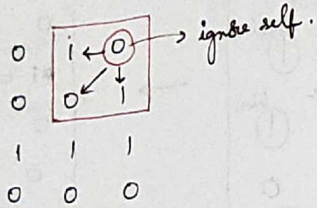


$(i,j) \rightarrow (9,2)$

row: -1 to 1 (0) — -1 invalid

col: 1 to 3 (2) — 3 invalid.

(0,1)
(0,2) → ignore self.
(1,1)
(1,2)



Steps :-

func → calculateNeighbours()

① For each element, calculate neighbours (nbrs) and according to the neighbour rule, update the element.

② As we need to preserve original values of the matrix to calculate neighbors of each element, the updated values obtained in step ① need to be stored in new matrix (temp)

③ Copy all elements from temp to original matrix.

Time complexity :- $O(mn)$.

We calculateNeighbors() for all mn elements.

calculateNeighbors() is a constant time method as it always counts atmost 8 neighbours

Space complexity :- $O(mn)$ → for temp array temp[m][n].

Program :-

```
int[][] temp = new int[m][n];

for(int i=0; i<m; i++) {
    for(int j=0; j<n; j++) {
        int noOfNbrs = calculateNeighbours(matrix, i, j);
        if( noOfNbrs < 2 || noOfNbrs > 3 ) {
            temp[i][j] = 0;
        } else if (noOfNbrs == 3 ) {
            temp[i][j] = 1;
        }
    }
}

calculateNeighbours ( int[][] matrix, int i, int j) {

    for(int row = Math.max(i-1, 0); row <= Math.min
        (i+1, m-1); row++) {
        for (int col = Math.max(j-1,0); col <= Math.min
            (j+1, n-1); col++) {
            sum += matrix[row][col];
        }
    }
    sum -= matrix[i][j];
    return sum;
}
```

Approach 2:- Optimal approach.

Update elements of matrix in place.

Time complexity: O(mn).

Space complexity: O(1).

beginning of revision

(In arrays, for problem: Rearrange Sorted Array in Max/Min form, we found a way to update array in-place using an encode-decode method.

Encoding Formula:

$$arr[i] += (arr[maxIdx] \% maxElem) * maxElem.$$

where $maxIdx = n-1$, $minIdx = 0$,

$maxElem = max(arr) + 1$

maxIdx for even positions
minIdx for odd positions

Decoding Formula:

$$arr[i] /= maxElem.$$

eg.,

arr:

3	4	1	2	6	5

sort →

	0	1	2	3	4	5
arr:	1	2	3	4	5	6

expected o/p →

	0	1	2	3	4	5
arr:	6	1	5	2	4	3

even pos. decreasing order (from largest element)
odd pos. increasing order (from smallest element)

⇒ 6 5 4

To avoid extra space, we save two values (i/p original value & o/p value) in the same position.

Meaning: if we want original i/p value at pos:0, we should fetch 1.

if we want o/p value at pos. 0, we should fetch 6.

We store 1 & 6 at same pos by encoding.

Illustration:

arr[0] = 1.

If we add 1 with a product of 7, then we can get 1 again by doing a modulo (%).

$arr[0] = 1 + (x) * 7$ } encoding.

say $x = 6$

$\therefore arr[0] = 1 + 6 * 7$

$= 43.$

$arr[0] \% 7 = 43 \% 7 = 1$ } decoding.

We can use "x" to store the o/p value (the second value).

$$\therefore x = \begin{cases} arr[maxIdx] & \text{for even} \\ arr[minIdx] & \text{for odd.} \end{cases}$$

But, wkt, when we fetch $arr[minIdx] = arr[0]$ while computing arr[1] (odd pos), that time arr[0] is already encoded as 43, so we need to do modulo (%) for original value

$\therefore x = \begin{cases} arr[maxIdx] \% maxElem & \text{for even index} \\ arr[minIdx] \% minElem & \text{for odd index} \end{cases}$

Hence $arr[i] = arr[i] + (arr[maxIdx] \% maxElem) * maxElem$ (for even pos)

$arr[i] = arr[i] + (arr[minIdx] \% maxElem) * maxElem$ (for odd pos)

) end of revision

In the current problem, we also need to store value in such a way that we obtain new value as well as old value.

We only have two bits : 0 & 1.

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

we can store the states as two bits

final state	initial state	
0	0	→ first 0 remains 0
0	1	→ first 1 becomes 0 dead
1	0	→ first 0 becomes 1 alive
1	1	→ first 1 becomes 1, remains alive

[<2 under population, >3 over population]

==3

==2 || ==3

00 → 0
01 → 1
10 → 2
11 → 3

To get initial state, x & 1

0	00 & 1 → 0
1	01 & 1 → 1
2	10 & 1 → 0
3	11 & 1 → 1

① $\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}$ nbrs = 1 (0) (00) → 0

② $\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}$ nbrs = 1 (0) dies. (01) → 1

③ $\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}$ nbrs = 2 (0) remains 0. (00) → 0

④ $\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}$ nbrs = 3 (1) alive (10) → 2

⑤ $\begin{bmatrix} 0 & 1 & 0 \\ 2 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}$ nbrs = 1+1+1+1+1 +(10&1) = 5+0 = 5 (0) dies (00) → 0

⑥ $\begin{bmatrix} 0 & 1 & 0 \\ 2 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}$ nbrs = 1+1+1 = 3 dies (11) → 3

⑦ $\begin{bmatrix} 0 & 1 & 0 \\ 2 & 0 & 3 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}$ nbrs = (2&1)+0+1+0+0 = 1 dies (01) → 1

⑧ $\begin{bmatrix} 0 & 1 & 0 \\ 2 & 0 & 3 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}$ nbrs = (2&1)+0+(3&1)+1+1+0+0+0 = 0+0+1+1+1+0+0+0 = 3 lives (11) → 3

(9)

$$\begin{bmatrix} 0 & 1 & 0 \\ 2 & 0 & 3 \\ 1 & 3 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

nbrs = 1+1 = 2
dies.
$(11)_2 \rightarrow (3)_{10}$

(10)

$$\begin{bmatrix} 0 & 1 & 0 \\ 2 & 0 & 3 \\ 1 & 3 & 3 \\ 0 & 0 & 0 \end{bmatrix}$$

nbrs = 1+1 = 2
(remains dead)
$(00)_2 \rightarrow 0$

(11)

$$\begin{bmatrix} 0 & 1 & 0 \\ 2 & 0 & 3 \\ 1 & 3 & 3 \\ 0 & 0 & 0 \end{bmatrix}$$

nbrs = 1+1+1 = 3
becomes alive
$(10)_2 \rightarrow (2)_{10}$

(12)

$$\begin{bmatrix} 0 & 1 & 0 \\ 2 & 0 & 3 \\ 1 & 3 & 3 \\ 0 & 2 & 0 \end{bmatrix}$$

nbrs = 1+1+0 = 2
remains dead.
$(00)_2 \rightarrow (0)_{10}$

$$\begin{bmatrix} 0 & 1 & 0 \\ 2 & 0 & 3 \\ 1 & 3 & 3 \\ 0 & 2 & 0 \end{bmatrix}$$

This is updated matrix with encoded values.

To obtain new states, we need to do "x >> 1".

	new	old			
0	(00)	⇒	00 >>1	⇒	0
1	(01)	⇒	01 >>1	⇒	0
2	(10)	⇒	10 >>1	⇒	1
3	(11)	⇒	11 >>1	⇒	1

∴ o/p matrix:

$$\begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Program :-

```
for (int i=0; i<m; i++) {
    for (int j=0; j<n; j++) {
        int nbrs = calculateNeighbours(matrix, i, j);
        if (matrix[i][j] == 1 && nbrs >= 2 && nbrs <= 3) {
            matrix[i][j] = 3;   // 01 → 11
        } else if (matrix[i][j] == 0 && nbrs == 3) {
            matrix[i][j] = 2;   // 00 → 10
        }
    }
}
for (int i=0; i<m; i++) {
    for (int j=0; j<n; j++) {
        matrix[i][j] >>= 1;
    }
}
int calculateNeighbours(int[][] matrix, int row, int col) {
    int neighbours = 0;
    for (int i = Math.max(row-1, 0); i <= Math.min(
            row+1, m-1); i++) {
        for (int j = Math.max(col-1, 0); j <= Math.min(
                col+1, n-1); j++) {
            neighbours += matrix[i][j] & 1;
        }
    }
    neighbours -= matrix[row][col];
    return neighbours;
}
```

Problem: Set Matrix Zeroes (LC73)

$$arr = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \rightarrow arr = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

do it "in-place"

Approach 1:-

$$arr = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

copy all elements to arr2

$$arr2 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

① traverse each elements of arr & when arr[i][j] = 0 then update all arr[i][:] = 0, arr2[:][j] = 0 — $O(mn) * O(m+n)$

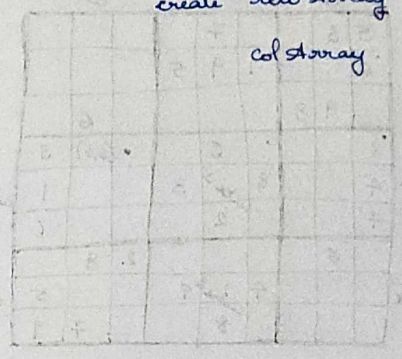
Time complexity: $O(mn * (m+n))$

$$arr2 = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 1 \end{bmatrix} \xrightarrow{copy} arr = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

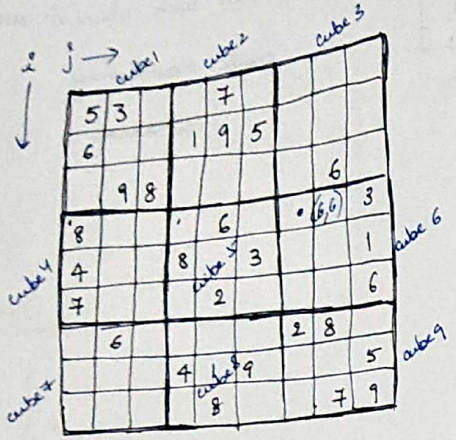
Approach 2:-

$$arr = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

① For each element arr[i][j], create rowArray, colArray



Leetcode 36: VALID SUDOKU



(0,0) → 5

rows : 5

cols : 5

cubes :

(0,0) = 5 belongs to first cube.

There are 9 cubes in cube array.

we determine indexes of 9 cubes as:

i: 0 to 2, j: 0 to 2 — cube 1 : (0,0) — (i,j) → (0,0) (0,1) (0,2) (1,0) (1,1) (1,2) (2,0) (2,1) (2,2)

i: 0 to 2, j: 3 to 5 — cube 2 : (0,1) — (i,j) → (0,3) (0,4) (0,5) (1,3) (1,4) (1,5) (2,3) (2,4) (2,5)

i: 0 to 2, j: 6 to 8 — cube 3 : (0,2) — (i,j) → (0,6) (0,7) (0,8) (1,6) (1,7) (1,8) (2,6) (2,7) (2,8)

i: 3 to 5, j: 0 to 2 — cube 4 : (1,0) — (i,j) → (3,0) (3,1) (3,2) (4,0) (4,1) (4,2) (5,0) (5,1) (5,2)

i: 3 to 5, j: 3 to 5 — cube 5 : (1,1)

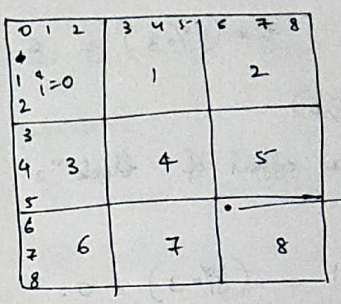
i: 3 to 5, j: 6 to 8 — cube 6 : (1,2)

cube 7 : (2,1)

cube 8 : (2,2)

cube 9 : (2,3)

Significance of i :- to determine (rowIdx, colIdx) of block.



i=8 means (6,6)

(6,6) is starting index of block 8.

i/3 — quotient for row

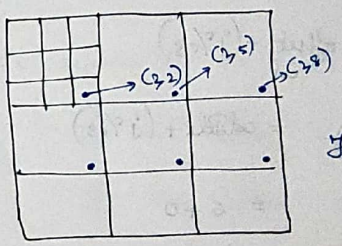
i%3 — remainder for col

3 * (i/3) = 6

3 * (i%3) = 6

∴ (rowIdx, colIdx) = (6,6)

Significance of j :-



j = 8 means (2,2).

There is similar pos like (2,2) for each block

→ (2,2)

(2,5)

(2,8)

∴ j=8 represents last cell of each block.

~~the exact position of~~

Significance of (i,j) together :- i = 8 j = 8.

⇒ (Last cell of last block)

(or)

(8,8) → (8th cell of 8th block)

8th cell of 8th block =

(i, j) = (2, 3) :-

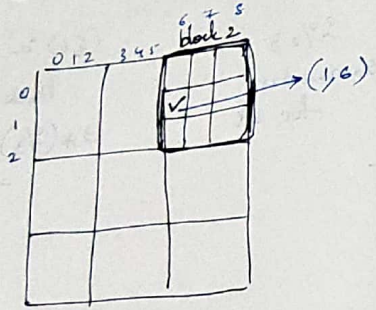
rowIdx = 3 * (2/3) = 0

colIdx = 3 * (2%3) = 6.

(0, 6)

This is offset of block "2".

(i/3) = 1 (j%3) = 0.



offset_row + (i/3) offset_col + (j%3)

= rowIdx + (i/3) = colIdx + (j%3)

= 0 + 1 = 6 + 0

= 1 = 6.

Approach :-

① Let 'i' be the outer loop.

It represents

① row in rowMap

② col in colMap

③ block$_{3\times3}$ in ~~(3×3) block~~. blockMap.

rowMap tracks elements in row i

colMap tracks elements in col i

blockMap tracks elements in block i.

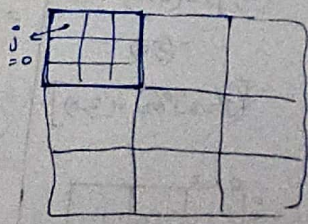
② Let 'j' be the variable in inner loop.

It represents

① col index in rowMap.

② row index in colMap

③ each cell in block.



③ For each (i, j)

check block[i][j] != '.'

block[j][i] != '.'

~~If both are~~ and store them in

rowMap

colMap.

④ Also for each (i,j);

~~for~~ ~~each~~

check each block i for j: 0 → 8.

By when j = 8, all cells in the i^{th} block are verified & each cell (if != '.') is added to block Map.

To do this, ~~for~~ ~~each block~~

Step (i): get the offset calculated with i.

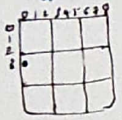
offset (rowIdx, colIdx)

= (3 * i/3, 3 * i%3)

This gives the first cell of the current block 'i'

e.g. if i = 3,

offset (3,0)



Step (ii): get the exact cell of the block with 'j' & offset.

(i', j')

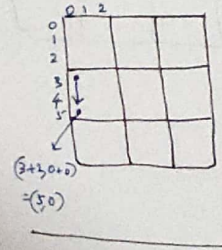
= offset + (j/2, j%2).

e.g. if j = 4,

(i', j') = (3+2, 0+0)

= (5,0)

∵ [(j/2, j%2) = (2,0)]



⑤ Thus, traverse (block-by-block) with 'i' & (cell-by-cell) with 'j' to add elements to blockMap along with

& rowMap, colMap

At any point, if the element is not being added to the Map, return false.

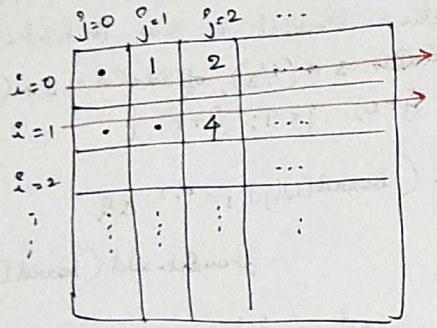
It is invalid Sudoku.

Program:-

```
for (int i=0; i<9; i++) {
    Set<Character> rowSet = new HashSet<>();
    Set<Character> colSet = new HashSet<>();
    Set<Character> blockSet = new HashSet<>();
    int offIdxRow = 3*(i/3), offIdxCol = 3*(i%3);
    for (int j=0; j<9; j++) {
        if (board[i][j] != '.' &&
                !rowSet.add(board[i][j])) {
            return false;
        }
        if (board[j][i] != '.' &&
                !colSet.add(board[j][i])) {
            return false;
        }
        int i' = offIdxRow + (j/3);
        int j' = offIdxCol + (j%3);
        if (board[i'][j'] != '.' &&
                !blockSet.add(board[i'][j'])) {
            return false;
        }
    }
}
return true;
```

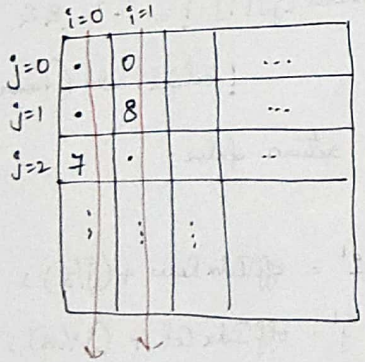
Diagramatic representation :-

① row Map — i represent row, j represent col.



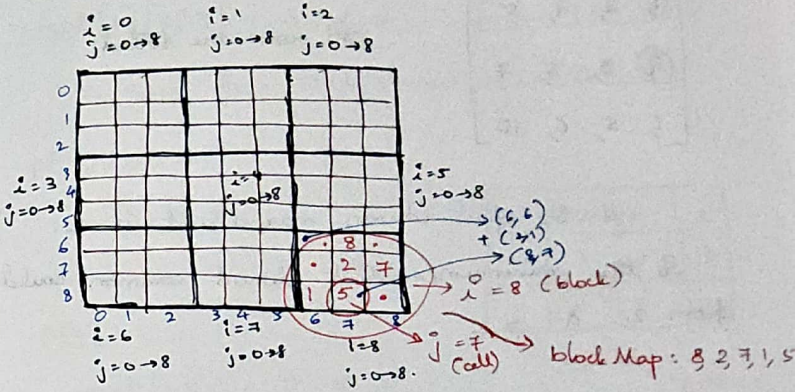
rowMap refreshes for every increment of 'i'

② colMap. — i represent col, j represent row.



colMap refreshes for every increment of 'i'.

③ block Map — i represent block, j represent each cell of block.



blockMap refreshes for every increment of

i: used to find offset (of first cell) of block i

$$\text{offset} = (3 * i/3,\ 3 * i\%3)$$

j: used to find the corresponding cell of block

$$(j/3,\ j\%3).$$

Finally $(i', j') = \text{offset} + (j/3,\ j\%3)$

Eg: In the above figure, the exact cell which has 5 is calculated as

for $(i,j) = (8,7)$; $(i', j') = \text{offset} + (j/3,\ j\%3)$

$= (3 * 8/3,\ 3 * 8\%3) + (7/3,\ 7\%3)$

$= (6,6) + (2,1)$

(6,6) ← offset of first cell

$= (8,7)$

LC1198 Smallest Common

$$\begin{bmatrix} 1, & 2, & 2, & 5 \\ (2), & 3, & 5, & 7 \\ 1, & 5, & 6, & 10 \end{bmatrix}$$

All rows are sorted.

In the first column, max element is 2.

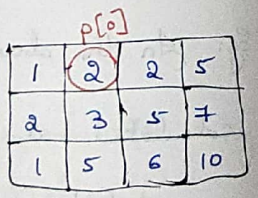
So the minimum/smallest element common could be starting from 2. & ≥ 2

p[i] is the colArray where it stores the column index of each row i, where max is found.

For each row, we update p[i] until it finds the max. If not found, i.e., p[i] == n we return -1; otherwise, we keep looking for the colIndex p[i] until value is $\geq$ max.

$\Rightarrow$ max = 2

row 0 :- p[0] = 0, mat[0][p[0]] = 1



$1 < 2$

$\therefore$ p[0]++

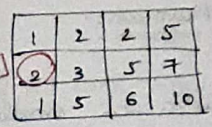
p[0] = 1, mat[0][1] = 2

$2 \neq 2$

$\therefore$ p[0] = 1

row 0 has max element at colIndex 1.

row 1 :- p[1] = 0.



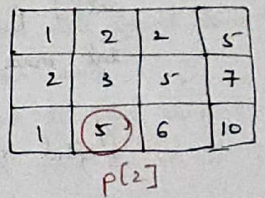
mat[1][p[1]] = 2. $\neq$ 2.

$\therefore$ row 1 has max element at colIndex 0.

row 2 :- p[2] = 0.

mat[2][0] = 1 < 2

p[2]++



$\therefore$ p[2] = 1

mat[2][1] = 5 $\neq$ 2

max = 5

$\therefore$ row 2 has max at colIndex 1

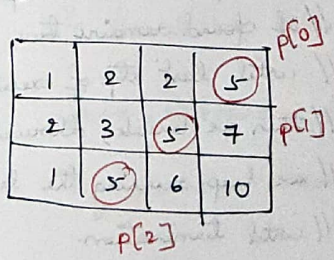
p	0	1	2
	1	0	1

But max element is now 5.

$\Rightarrow$ max = 5.

Repeat the same process until you find all p[i] pointing to 5.

Next steps :-



max = 5.

p	0	1	2
	3	2	1

Program:-

```
public int smallestCommonElement (int[][] mat) {
    int m = mat.length;
    int n = mat[0].length;
    int[] p = new int[m]; // pointer to the colums of each row.
    int max = mat[0][0];

    //find maximum element in first column
    for(int[] a : mat) {
        max = Math.max(max, a[0]);
    }

    // An infinite loop
    // In other words, a simulation that runs until an answer
    // is found & terminates when answer doesn't exist.
    while (true) {
        int nextMax = 0; // purpose: holds max value of elements at
                         // current positions of all row ptrs p[i]
        boolean found = true; // if found remains true
                              // until last step of execution,
                              // then we break, otherwise,
                              // we keep running the loop
                              // until termination.
        for (int i = 0; i < m; i++) {  // for each row
            // move col until we find next max element
            while (p[i] < n && mat[i][p[i]] < max) {
                p[i]++;
                found = false; // since we still found element
                               // less than max
                               // and yet to find element at least
                               // equal to max, we set it to
                               // false.
                               // if we found element which is
                               // mat[i][p[i]] >= max, then
                               // "found" flag remains true
            }

            if (p[i] == n) return -1; // when we exhausted
                                      // elements in row, meaning no
                                      // answer, so return -1

            nextMax = Math.max(nextMax, mat[i][p[i]]);
        } // nextMax holds ideal candidates for being smallest common element

        max = nextMax; // next max element updated

        if (found) break; // if all row elements were >= previous
                          // max, we can break out of loop.

    } // end of while loop.
    return max;
}
```

Dry Run :-

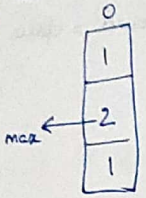
$$\begin{bmatrix} 1 & 2 & 2 & 5 \\ 2 & 3 & 5 & 7 \\ 1 & 5 & 6 & 10 \end{bmatrix}$$

p = | 0 | 0 | 0 |

ITERATION 1 :-

① * First iteration (finding initial max).

max = 2



② * Outer loop :-
nextMax = 0

found = true

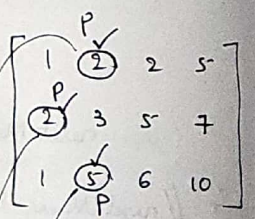
③ Inner loop :- found = false.

row 0 : mat[0,0] = 1 < 2 (max) → mat[0,1] = 2 = 2

row 1 : mat[1,0] = 2 = 2, found remains false.

row 2 : mat[2,0] = 1 < 2 (max), found = false, → mat[2,1] = 5,

p | 1 | 0 | 1 |



nextMax :

row 0 : nextMax = max (0, 2) = 2

row 1 : nextMax = max (2, 2) = 2

row 2 : nextMax = max (2, 5) = 5 ✓✓

nextMax = max (2, 2, 5) = 5.

④ max = nextMax

⇒ max = 5.

ITERATION 2 :-

① Outer loop :-

nextMax = 0

found = true.

② Inner loop :-

p : | ✗ | ✗ | 1 |
3 2

row 0 : mat[0][1] = 2 < 5 (max) so p[0]++ found = false

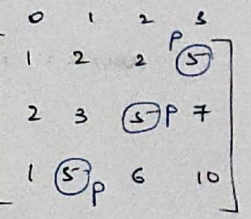
mat[0][2] = 2 < 5 (max)

p[0]++

found false.

mat[0][3] = 5 = = 5

found remains false. & p[0] remains 3.



row 1 : mat[1][0] = 2 < 5 (max) so p[1]++ found = false.

mat[1][1] = 3 < 5 (max) so p[1]++ found = false

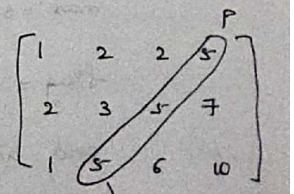
mat[1][2] = 5 = = 5 so found remains false &

p[1] remains 2.

row 2 : mat[2][1] = 5 = = 5 (max) found remains false &

p[2] remains 1.

∴ p | 3 | 2 | 1 |



nextMax :

row 0 : nextMax = max (0, 5) = 5

row 1 : nextMax = max (5, 5) = 5

row 2 : nextMax = max (5, 5) = 5.

nextMax = max (5, 5, 5) = 5

③ max = nextMax

⇒ max = 5

ITERATION 3 :-

① Outerloop : nextMax = 0, found flag = true.

② Innerloop :

	0	1	2
p	3	2	1

row 0: mat[0][3] = 5 == 5, found remains true & p[0] remains 3.

row 1: mat[1][2] = 5 == 5, found remains true & p[1] remains 2.

row 2: mat[2][1] = 5 == 5, found remains true & p[2] remains 1

	0	1	2
p	3	2	1

p

1	2	2	5
2	3	5	7
1	5	6	10

nextMax :

row 0: nextMax = max(0, 5) = 5

row 1: nextMax = max(5, 5) = 5

row 2: nextMax = max(5, 5) = 5

③ max = nextMax

max = 5.

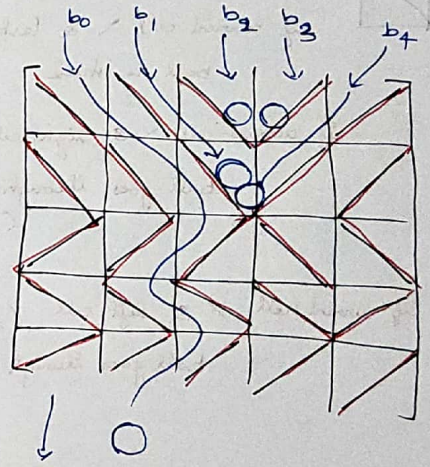
flag = = true, to break out of loop and return max = 5.

LC 1706: WHERE WILL THE BALL FALL

$$\begin{bmatrix} 1 & 1 & 1 & -1 & -1 \\ 1 & 1 & 1 & -1 & -1 \\ -1 & -1 & -1 & 1 & 1 \\ 1 & 1 & 1 & 1 & -1 \\ -1 & -1 & -1 & -1 & -1 \end{bmatrix}_{m \times n}$$

n balls.

represented as



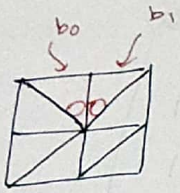
ans :-

b_0	b_1	b_2	b_3	b_4
1	-1	-1	-1	-1

Means, b_0 comes out of column "1"

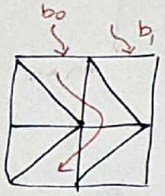
b_1, b_2, b_3, b_4 are all "-1" because they are stuck.

Pattern :-



b_0, b_1 are stuck.

if current cell \, right cell /
ball is stuck
and viceversa.



b_0, b_1

if current cell \ & last column,
ball is stuck

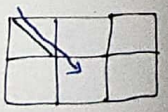
if current cell \ & right cell \,
ball goes through.
(i+1, j+1)

See more such patterns in "termination conditions"

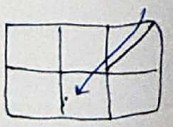
if current cell / & left cell /,
ball goes through (i+1, j-1).

Intuition :-

If ball starts from column 0, row 0 and value in the cell grid[0][0] is 1 (\) the ball will move to next row, which is 1 & next column, which is also 1.
So next position of the ball will be grid[1][1].



If ball starts from column 2, row 0 and value in the cell grid[0][2] is -1 (/) the ball will move to next row, which is 1 & previous column, which is 1.
So next position of the ball will be grid[1][1]

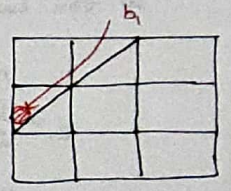


Termination conditions :-

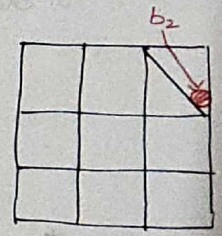
① Ball keeps moving to next rows until it hits last row in the grid.
This is a success case, and the column where the ball reach the final row is recorded in the output array.

② The ball gets stuck in a row because one of the following conditions is met:

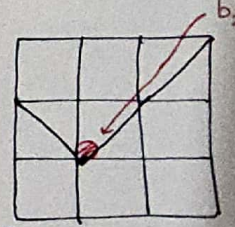
(i) ball falls to leftmost column & value is -1 (/)



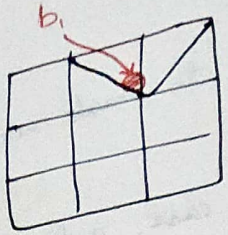
(ii) ball falls to right most column & value is 1 (\)



(iii) ball is between two opposite diagonals -1, 1.
Current cell is \ (1), previous column has / (-1)
(forming V shape)



(iv) ball is between two opposite diagonals 1, -1
Current cell is \ (1), immediate next column has / (-1)
(forming V shape).



Iterative approach:-

① Iterate the grid using nested for.

For each column, perform the following:

② For each 'row', do the following:

1. nextCol = currCol + grid[row][currCol].
2. check if grid[row][nextCol] == grid[row][currCol]

   If so, ball can move to next row.

3. check if ball is not stuck

   nextCol < 0
   (or)
   nextCol > n-1
   } => ball is stuck.

4. set currCol = nextCol.
5. If this is last row i.e., row == m-1,
   result[col] = nextCol.