Matrix or 2D array or Grid

0-based indexing

	C0	C1	C2	C3	C4
R0	0,0	0,1	0,2	0,3	0,4
R1	1,0	1,1	1,2	1,3	1,4
R2	2,0	2,1	2,2	2,3	2,4
R3	3,0	3,1	3,2	3,3	3,4
R4	4,0	4,1	4,2	4,3	4,4
R5	5,0	5,1	5,2	5,3	5,4

$m \times n$

6×5

Derived datatype

```
int [][] mat
    = new int [6][5];

for(int row = 0; row < 6; row++){
    for(int col = 0; col < 5; col++){
        mat [row][col] = sc.nextInt();
        Syso(mat [row][ col] + " ");
    }
    Syso.println();
}
```

```java
int rows = 6, cols = 5;
int[][] mat = new int[rows][cols];

// Input
for(int row = 0; row < rows; row++){
    for(int col = 0; col < cols; col++){
        mat[row][col] = row * 10 + col;
    }
}

// Output
for(int row = 0; row < rows; row++){
    for(int col = 0; col < cols; col++){
        System.out.print(mat[row][col] + " ");
    }
    System.out.println();
}
```

not preferred

no of cells = N (eg 30)

Time ⇒ O(N)

Space (Create matrix) ⇒ O(N)

no of rows = m (6), no of cols = n (5)

Time ⇒ O(m x n)

Input Space (matrix) ⇒ O(m x n)

quadratic

memory mapping

C++ (2d arrays → row major form)

	c0	c1	c2	c3	c4
r0	0,0	0,1	0,2	0,3	0,4
r1	1,0	1,1	1,2	1,3	1,4
r2	2,0	2,1	2,2	2,3	2,4
r3	3,0	3,1	3,2	3,3	3,4
r4	4,0	4,1	4,2	4,3	4,4
r5	5,0	5,1	5,2	5,3	5,4

1d array

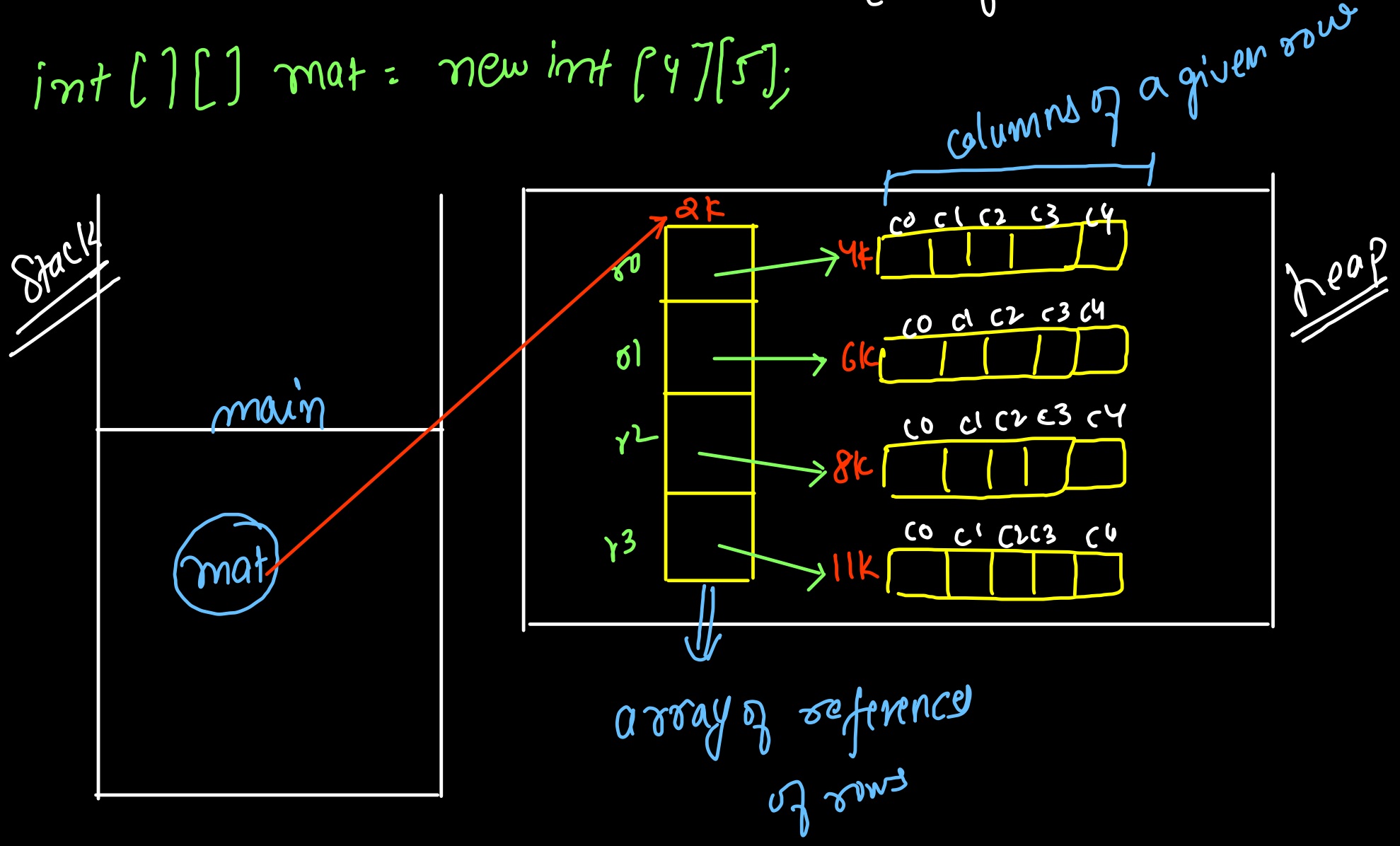
advantage: contiguous (random access)

disadvantage: availability ↳ contiguous space

Java (jagged arrays)

{ no of rows = mat.length
  no of columns = mat[0].length }

```
int [ ][ ] mat = new int [4][5];
```



```java
Scanner scn = new Scanner(System.in);

int rows = scn.nextInt();
int cols = scn.nextInt();
int[][] mat = new int[rows][cols];

// Input
for(int row = 0; row < mat.length; row++){
    for(int col = 0; col < mat[0].length; col++){
        mat[row][col] = scn.nextInt();
    }
}

// Output
for(int row = 0; row < mat.length; row++){
    for(int col = 0; col < mat[0].length; col++){
        System.out.print(mat[row][col] + " ");
    }
    System.out.println();
}
```

```
Finished in 127 ms
0 1 2 3
10 11 12 13
20 21 22 23
```

stdin

```
3 4
00 01 02 03
10 11 12 13
20 21 22 23
```

```java
int[][] mat = new int[4][];

for(int row = 0; row < mat.length; row++){
    mat[row] = new int[row + 1];
    for(int col = 0; col < mat[row].length; col++){
        mat[row][col] = row * 10 + col;
        System.out.print(mat[row][col] + " ");
    }
    System.out.println();
}
```

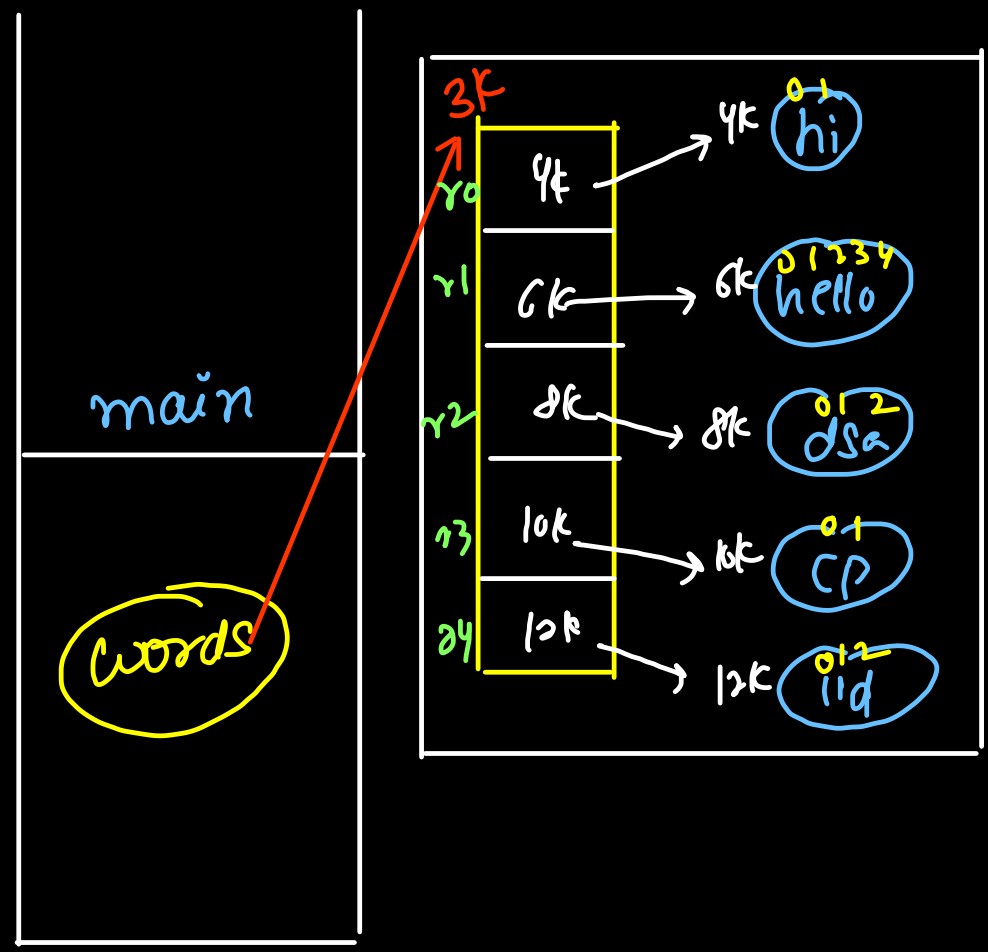
```
Finished in 138 ms
0
10 11
20 21 22
30 31 32 33
```

Each row can have different no of columns

```java
String[] words = new String[5];

for(int idx = 0; idx < words.length; idx++){
    words[idx] = scn.next();
    System.out.print(words[idx] + " ");
}

System.out.println();

for(int row = 0; row < words.length; row++){
    for(int col = 0; col < words[row].length(); col++){
        System.out.print(words[row].charAt(col) + " ");
    }
    System.out.println();
}
```

→ string fix
→ character

```
h i
h e l l o
d s a
c p
l l d
```



→ List< List<?> >
→ methods → matrix
postreturn
→ shallow vs deep

	c0	c1	c2	c3	c4
r0	0,0	0,1	0,2	0,3	0,4
r1	1,0	1,1	1,2	1,3	1,4
r2	2,0	2,1	2,2	2,3	2,4
r3	3,0	3,1	3,2	3,3	3,4
r4	4,0	4,1	4,2	4,3	4,4
r5	5,0	5,1	5,2	5,3	5,4

```java
// Output: Row by Row Traversal
for(int row = 0; row < mat.length; row++){
    for(int col = 0; col < mat[0].length; col++){
        System.out.print(mat[row][col] + " ");
    }
    System.out.println();
}
System.out.println();

// Output: Col by Col Traversal
for(int col = 0; col < mat[0].length; col++){
    for(int row = 0; row < mat.length; row++){
        System.out.print(mat[row][col] + " ");
    }
    System.out.println();
}
```

O(m) O(n) O(1) O(mxn) quadratic

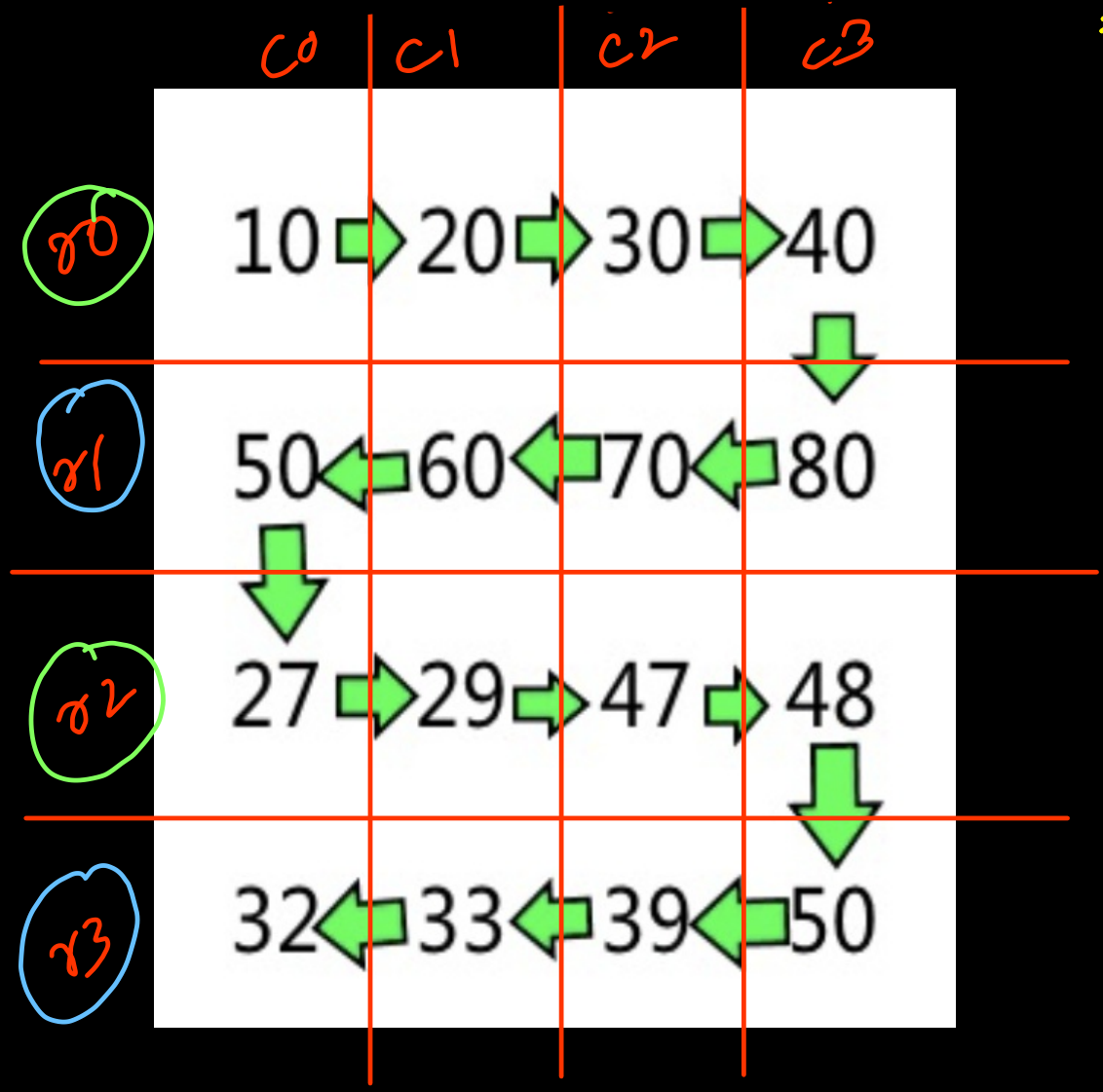
O(n) O(m) O(1) O(nxm) quadratic

how level

row by row ⇒ better ⇒ locality of reference

GFG

Snake Traversal



row by row ⇒ outer loop

col by col ⇒ inner loop

↳ odd rows ⇒ right to left
(last col to first col)

even rows ⇒ left to right
(first col to first col)

```java
static ArrayList<Integer> snakePattern(int[][] mat)
{
    ArrayList<Integer> res = new ArrayList<>();

    // top to down row by row
    for(int row = 0; row < mat.length; row++){
        if(row % 2 == 0){
            // even row -> left to right
            for(int col = 0; col < mat[0].length; col++){
                res.add(mat[row][col]);
            }
        } else {
            // odd row -> right to left
            for(int col = mat[0].length - 1; col >= 0; col--){
                res.add(mat[row][col]);
            }
        }
    }

    return res;
}
```

output space: $O(n^2)$

input space: $O(n^2)$

$O(rows)$

$O(cols)$

Time # rows = cols = n

$O(rows * cols)$

quadratic

Space

$O(1)$ constant (inplace)

extra space

Convert 1D to 2D array

10	20	30	40	50	60	70	80	90	100	110	120
0	1	2	3	4	5	6	7	8	9	10	11

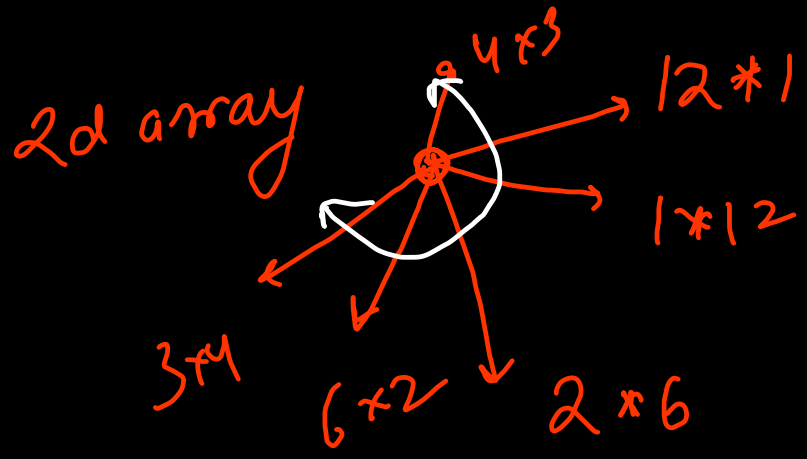
idx

n1 = 12

3 * 4

rows ↑ Cols ↑

1) convert



	c0	c1	c2	c3
r0	10	20	30	40
r1	50	60	70	80
r2	90	100	110	120

3*4

```java
public int[][] construct2DArray(int[] original, int m, int n) {
    // conversion not possible
    if(m * n != original.length){
        return new int[0][0];
    }

    int[][] mat = new int[m][n];

    int idx = 0;
    for(int row = 0; row < m; row++){
        for(int col = 0; col < n; col++){
            mat[row][col] = original[idx++];
        }
    }

    return mat;
}
```

output

input

↳ N elements

// empty array → m x n = 1x1 ≠ 4, m x n = 2x3 ≠ 4

① store
② idx++

approach ①

idx

0	1	2	3
10	20	30	40

n = 4

	c0	c1
r0	10	20
r1	30	40

m x n
2 x 2

Time = linear O(N) = O(m x n)
N = no. of elements in array
m x n = no. of matrix cells

Space = O(1) constant/inplace

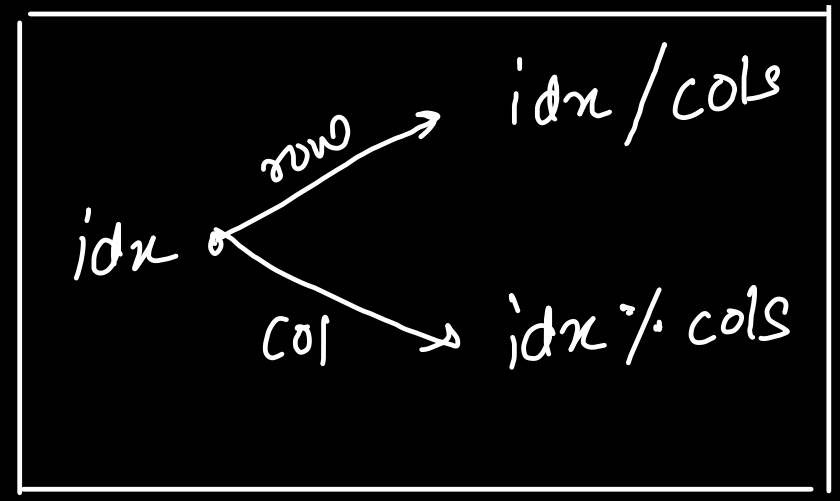
2nd approach

10	20	30	40	50	60	70	80	90	100	110	120
0	1	2	3	4	5	6	7	8	9	10	11

	c0	c1	c2	c3
r0	10	20	30	40
r1	50	60	70	80
r2	90	100	110	120

rows, cols
3 * 4

0 → (0,0) 0/4 0%4
1 → (0,1) 1/4 1%4
2 → (0,2) 2/4 2%4
3 → (0,3) 3/4 3%4
4 → (1,0) 4/4 4%4
5 → (1,1) 5/4 5%4
6 → (1,2) 6/4 6%4
7 → (1,3) 7/4 7%4
8 → (2,0) 8/4 8%4
9 → (2,1) 9/4 9%4
10 → (2,2) 10/4 10%4
11 → (2,3) 11/4 11%4



```java
public int[][] construct2DArray(int[] original, int m, int n) {
    // conversion not possible
    if(m * n != original.length){
        return new int[0][0];
    }

    int[][] mat = new int[m][n];

    for(int idx = 0; idx < original.length; idx++){
        int row = idx / n;
        int col = idx % n;
        mat[row][col] = original[idx];
    }

    return mat;
}
```

LC 2022

row = idx / cols

col = idx % cols

/* formula to convert 1D to 2D */

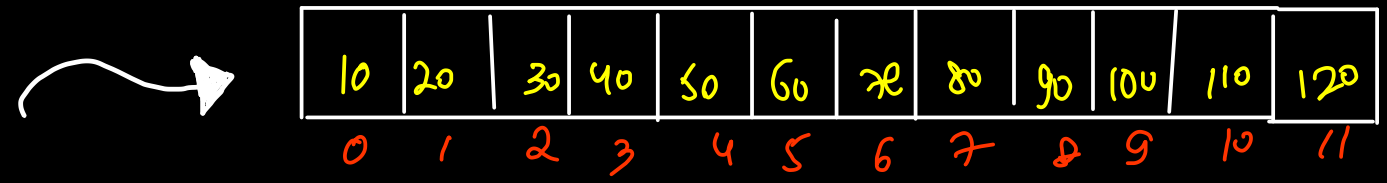
Time ⇒ O(m x n) = O(N) linear

Space ⇒ O(1) inplace

* ⇒ m = no of rows
* ⇒ n = no of columns
* ⇒ N = 1D array length = m x n

Conversion from 2D matrix to 1D array

	c0	c1	c2	c3
r0	10	20	30	40
r1	50	60	70	80
r2	90	100	110	120



3 * 4

m x n

idx = (r * cols + c)

2d coordinates → 1d coordinate

(0,0) → 0 (0×4+0)

(0,1) → 1 (0×4+1) col ⇒ +1

(0,2) → 2 (0×4+2) +4

(0,3) → 3 (0×4+3)

(1,0) → 4 (1×4+0)

(1,1) → 5 (1×4+1) +4

(1,2) → 6 (1×4+2) +4

(1,3) → 7 (1×4+3) row ⇒ +4

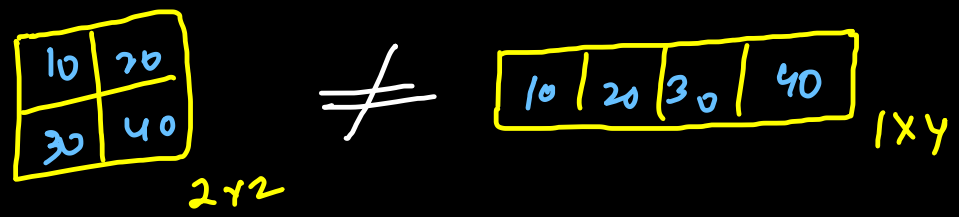
(2,0) → 8 (2×4+0)

(2,1) → 9 (2×4+1)

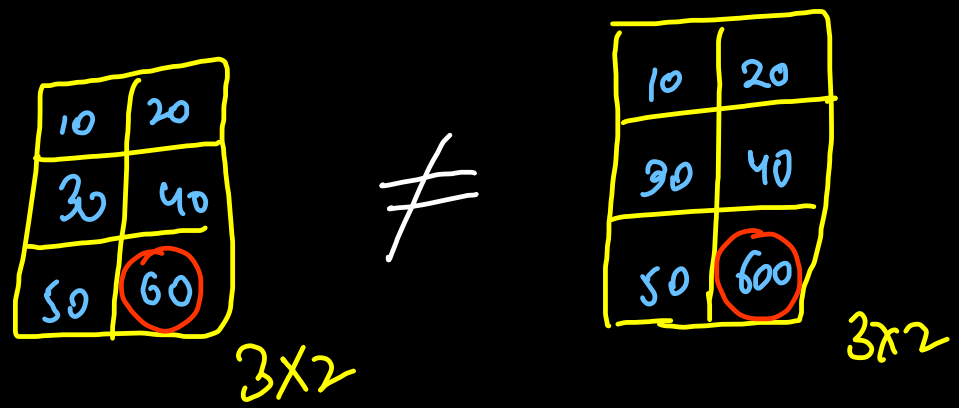
(2,2) → 10 (2×4+2)

(2,3) → 11 (2×4+3)

Identical matrices



① $rows_1 \neq rows_2$ or $cols_1 \neq cols_2$ } non-identical



② any one value is different ⇒ non-identical

```
int areMatricesIdentical(int N, int[][] g1, int[][] g2) {
    if(g1.length != g2.length) {
        return 0; // rows are not same
    }
    if(g1[0].length != g2[0].length){
        return 0; // cols are not same
    }

    // values comparison
    for(int row = 0; row < g1.length; row++){
        for(int col = 0; col < g1[0].length; col++){
            if(g1[row][col] != g2[row][col]) {
                return 0;
            }
        }
    }

    return 1;
}
```

m x n

Time
↳ O(m x n)
= O(n²)
quadratic

Space
↳ O(1) constant
in place

Add 2 matrices

	c0	c1	c2	c3
r0	10	20	30	40
r1	50	60	70	80
r2	90	100	110	120

3X4

+

	c0	c1	c2	c3
r0	100	200	300	400
r1	500	600	700	800
r2	900	1000	1100	1200

3X4

$res(i,j) = m_1(i,j) + m_2(i,j)$

$m_1[i][j] += m_2[i][j]$

	c0	c1	c2	c3
r0	110	220	330	440
r1	550	660	770	880
r2	990	1100	1210	1320

3X4

```java
public void  Addition(int[][] m1, int[][] m2)
{
    for(int row = 0; row < m1.length; row++){
        for(int col = 0; col < m1[0].length; col++){
            m1[row][col] = m1[row][col] + m2[row][col];
        }
    }

}
```

Time = $O(n^2)$

Space = $O(1)$
inplace

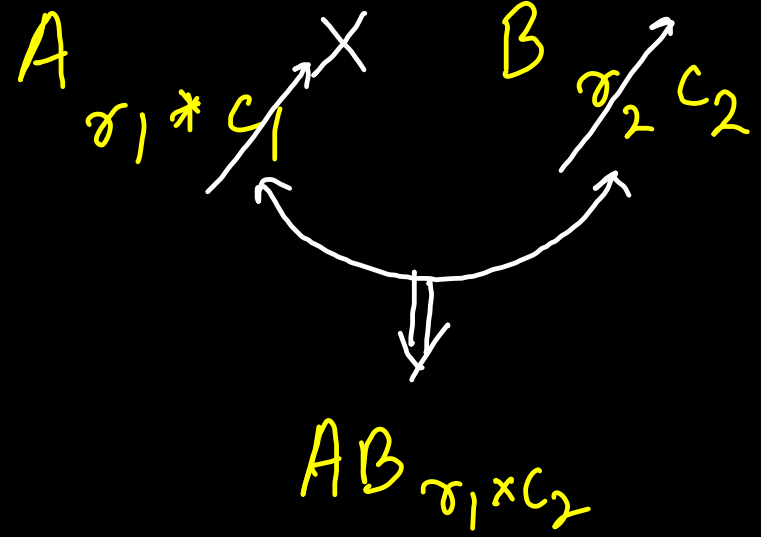
Subtract ⇒ replace + with −

multiplicat^n ⇒ this logic will fail

Matrix Multiplication

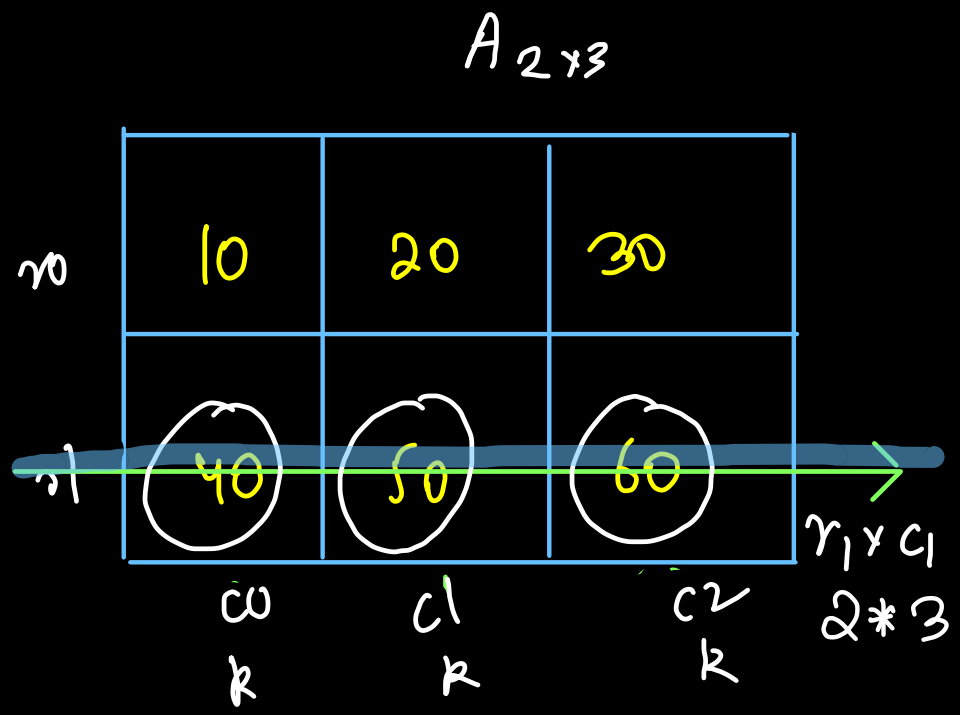
$A \times B \neq B \times A$

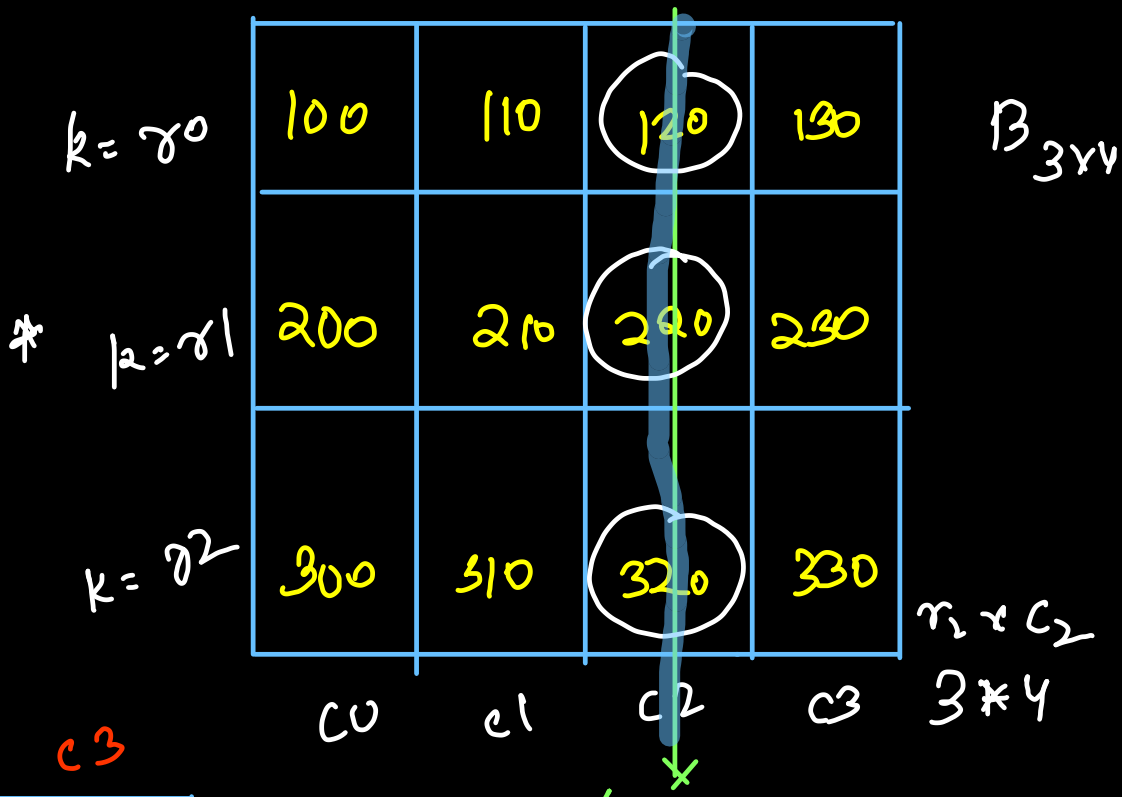
Commutative ✗



constraints $\Rightarrow c_1 = r_2$

Matrix multiplicatn possible only if constraint is true





3 ~~X~~ nested loops

$C_{2\times4}$

	c0	c1	c2	c3
r0	10x100 + 20x200 + 30x300	10x110 + 20x210 + 30x310	10x120 + 20x220 + 30x320	10x130 + 20x230 + 30x330
r1	A_{r_1} * B_{c0}	A_{r_1} * B_{c1}	A_{r_1} * B_{c_2}	A_{r_1} * B_{c3}

2x4
$r_1 \times c_2$

$C[r_1][c_2]$

$= A[r_1][0] * B[0][c_2]$
$+ A[r_1][1] * B[1][c_2]$
$+ A[r_1][2] * B[2][c_2]$

```java
public static void multiply(int A[][], int B[][], int C[][], int N)
    {
        int r1 = A.length, c1 = A[0].length;
        int r2 = B.length, c2 = B[0].length;

        if(c1 != r2) return; // matrix multiplication not possible

        // resultant dimensions: (r1, c2)

        for(int row = 0; row < r1; row++){
            for(int col = 0; col < c2; col++){
                for(int k = 0; k < c1; k++){
                    C[row][col] += A[row][k] * B[k][col];
                }
            }
        }
    }
```

input

output

$r_1 = c_1 = r_2 = c_2 = N$

Time $\Rightarrow O(N^3)$ cubic

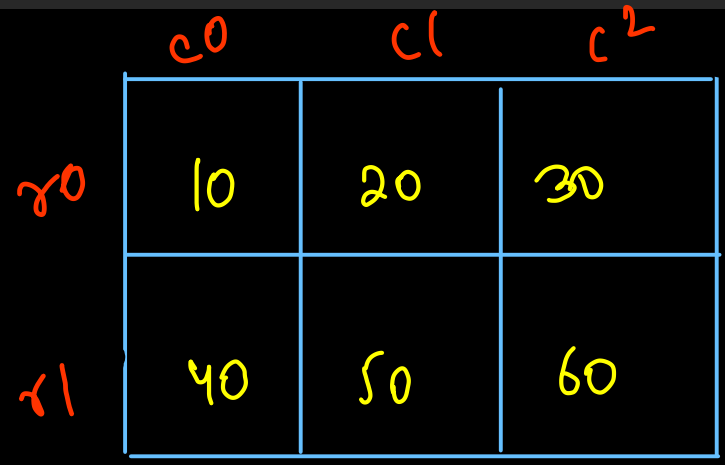
Space $\Rightarrow O(1)$ inplace

LC 867) Rectangular matrix Transpose

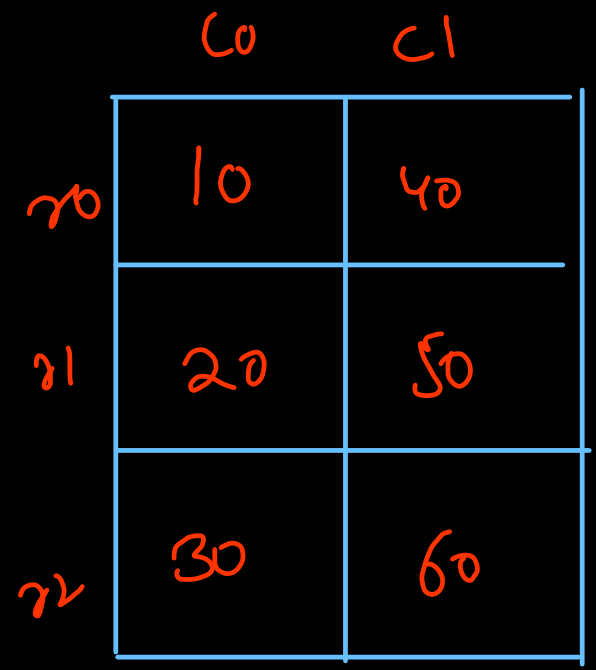
```java
public int[][] transpose(int[][] mat) {
    int m = mat.length, n = mat[0].length;
    int[][] res = new int[n][m];
            ↳ output
    for(int row = 0; row < res.length; row++){
        for(int col = 0; col < res[0].length; col++){
            res[row][col] = mat[col][row];
        }
    }

    return res;
}
```

Time = $O(n^2)$ quadratic

Space = $O(1)$ no extra space



2*3

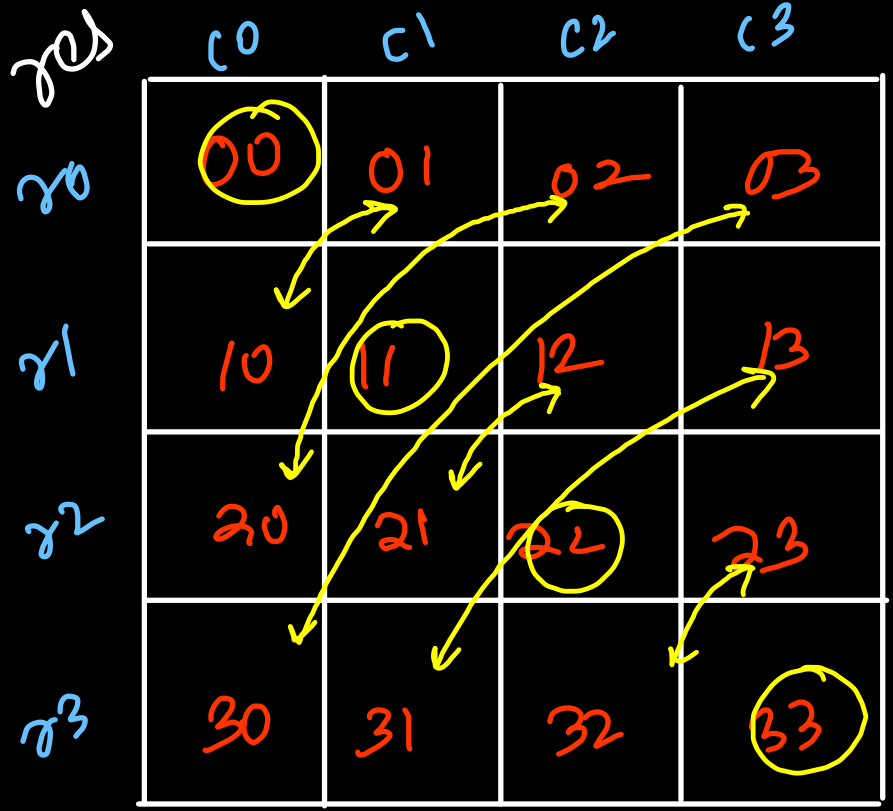


3*2

(GFG)

Tranpose Matrix → rows interchange with cols

↳ square matrix, new matrix ✗ not allowed.

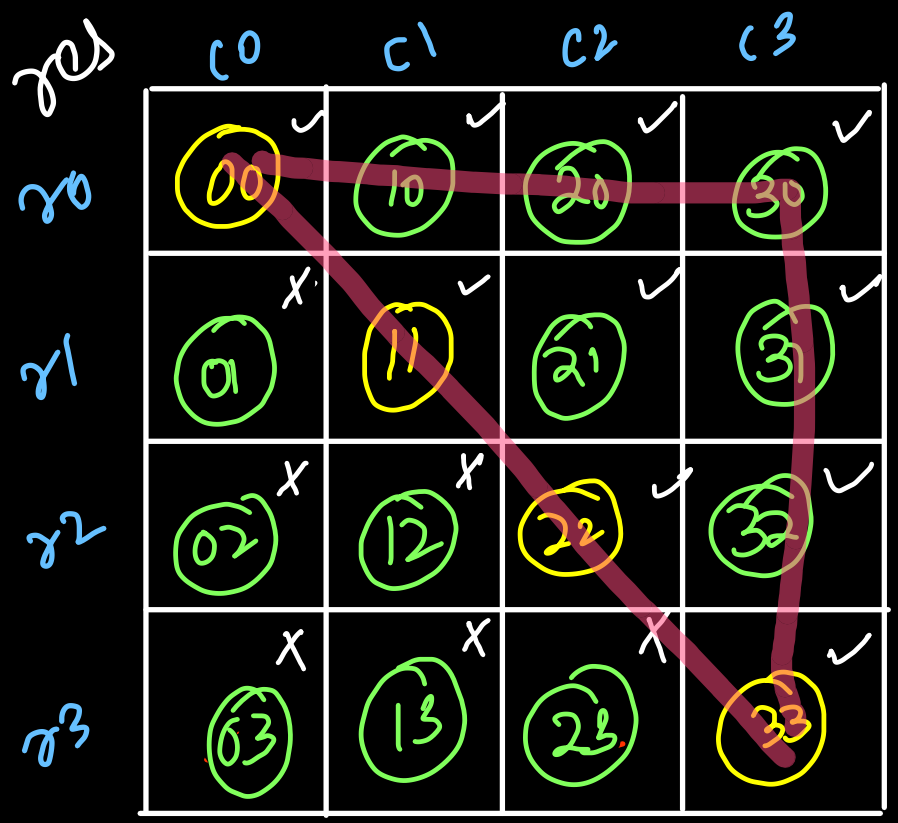


row → col

col → row

mat[r][c] ⇔ mat[c][r]

swap



$m(r, c) \hookrightarrow m(c, r)$

```
for(int r = 0; r < N; r++){
    for(int c = r; c < N; c++){
        swap(mat, r, c, c, r)
    }
}
```

output => Input dimensions

```
//Function to find transpose of a matrix.
static void transpose(int mat[][], int n)
{
    // run loop only in upper right triangle
    for(int row = 0; row < n; row++){
        for(int col = row; col < n; col++){
            // Swapping values at (row, col) with (col, row)
            int temp = mat[row][col];
            mat[row][col] = mat[col][row];
            mat[col][row] = temp;
        }
    }
}
```

Time => $O(n^2)$ quadratic

Space => O(1) inplace (w/o new matrix)

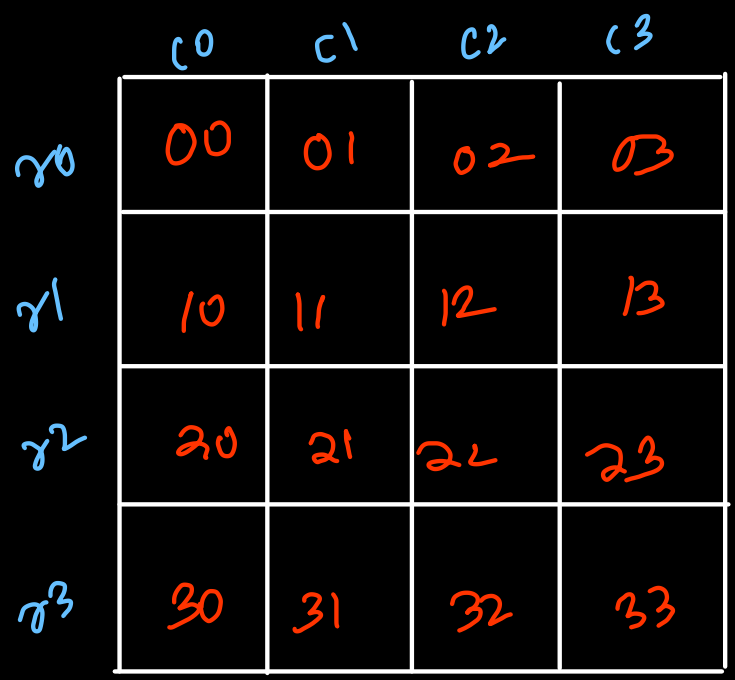
Leetcode Code

If I had created a new resultant matrix, then due to void return type, it will be considered $O(n^2)$ extra space

Rotate Image

Square matrix

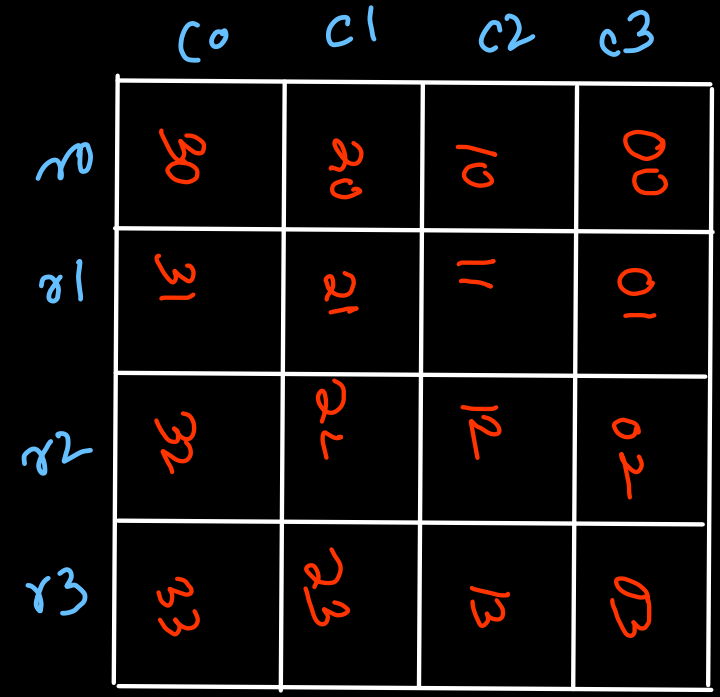
90° clockwise

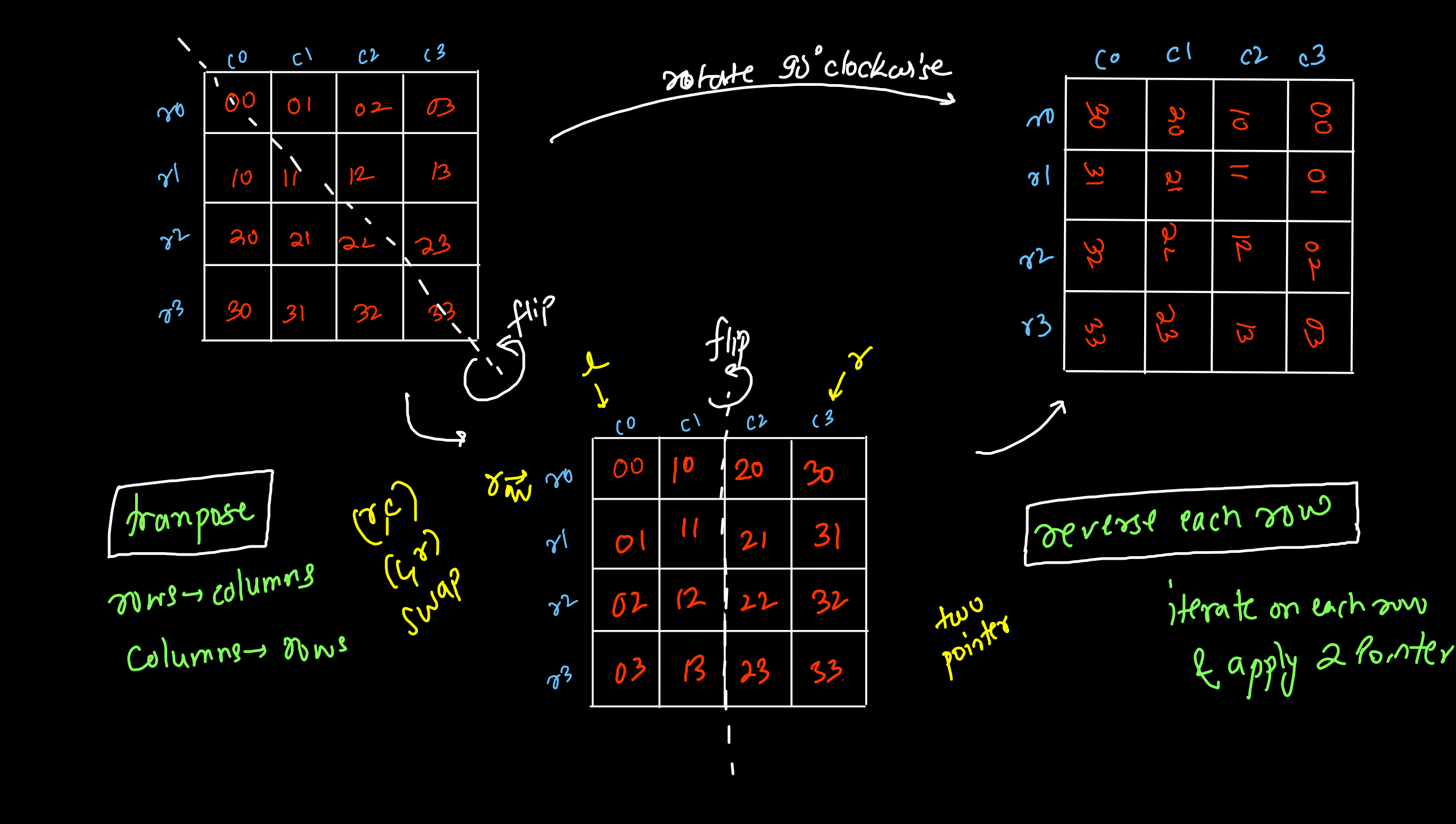


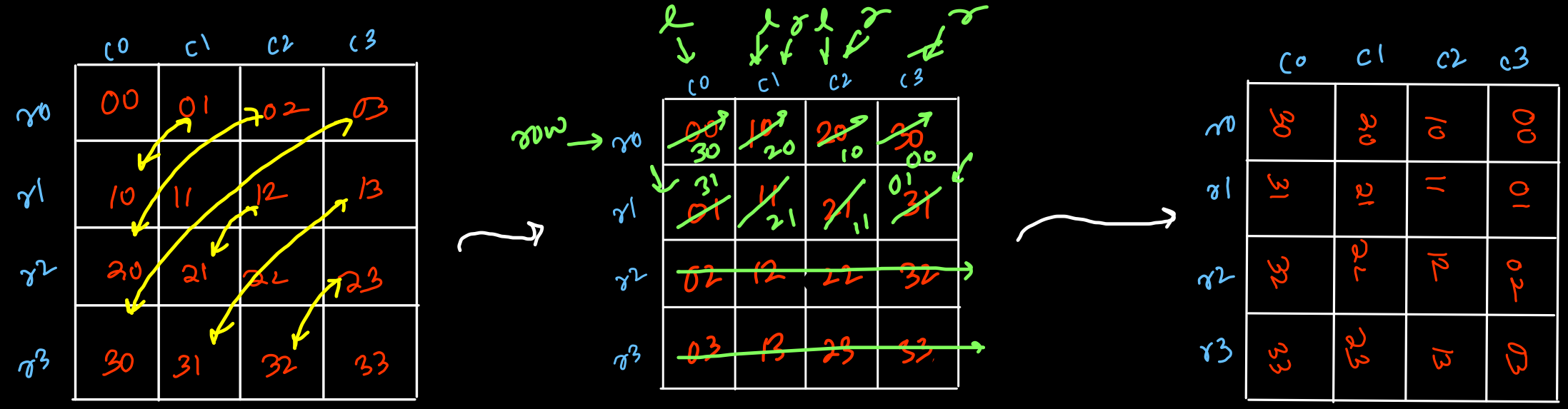
Inplace

think?

2 traversals




c0 c1 c2 c3
r0 00 01 02 03
r1 10 11 12 13
r2 20 21 22 23
r3 30 31 32 33
flip
flip
rotate 90° clockwise
c0 c1 c2 c3
r0 30 20 10 00
r1 31 21 11 01
r2 32 22 12 02
r3 33 23 13 03
c0 c1 c2 c3
r0 00 10 20 30
r1 01 11 21 31
r2 02 12 22 32
r3 03 13 23 33
transpose
rows → columns
Columns → rows
(r,c)
(c,r)
swap
two pointer
reverse each row
iterate on each row
& apply 2 pointer




```java
// swap (r1, c1) with (r2, c2) values
public static void swap(int[][] mat, int r1, int c1, int r2, int c2){
    int temp = mat[r1][c1];
    mat[r1][c1] = mat[r2][c2];
    mat[r2][c2] = temp;
}
```

```java
// rows -> columns, columns -> rows
public void transpose(int[][] mat, int n) {
    // run loop only in upper right triangle
    for(int row = 0; row < n; row++){
        for(int col = row; col < n; col++){
            // Swapping values at (row, col) with (col, row)
            swap(mat, row, col, col, row);
        }
    }
}
```

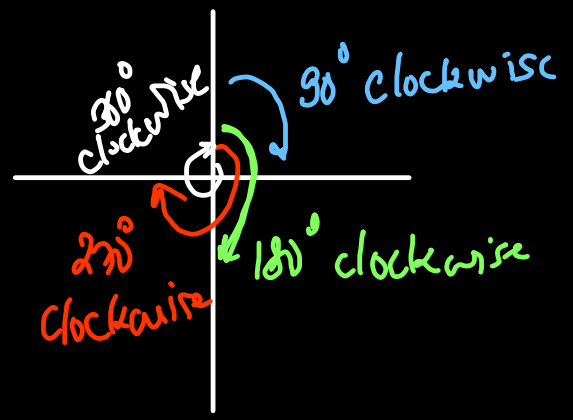
$O(n^2)$ time

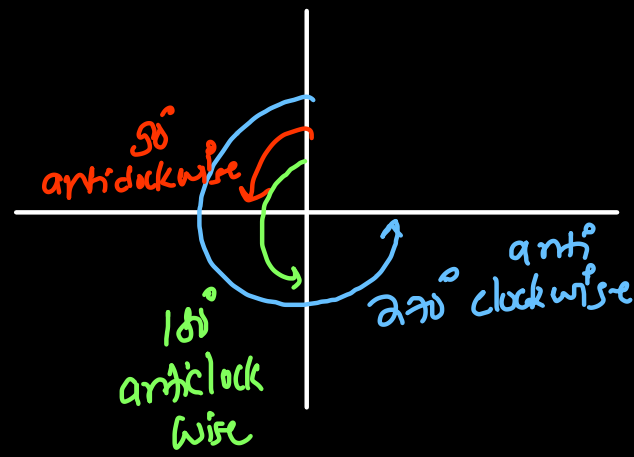
```java
// reverse each row by using two pointer technique
public void reverseRows(int[][] mat, int n){
    for(int row = 0; row < n; row++){
        int left = 0, right = n - 1;
        while(left < right){
            // swapping values at (row, left) with (row, right)
            swap(mat, row, left, row, right);
            left++; right--;
        }
    }
}

public void rotate(int[][] mat) {
    int n = mat.length;
    transpose(mat, n);
    reverseRows(mat, n);
}
```

$O(n^2)$ time

Time $\Rightarrow O(n^2)$

Space $\Rightarrow O(1)$ inplace





hw

	c0	c1	c2	c3
r0	00 03	01 02	02 02	03 00
r1	10 13	11 12	12 11	13 10
r2	20 23	21 22	22 21	23 20
r3	30 33	31 32	32 31	33 30

(1) reverse rows

(2) reverse cols

180°

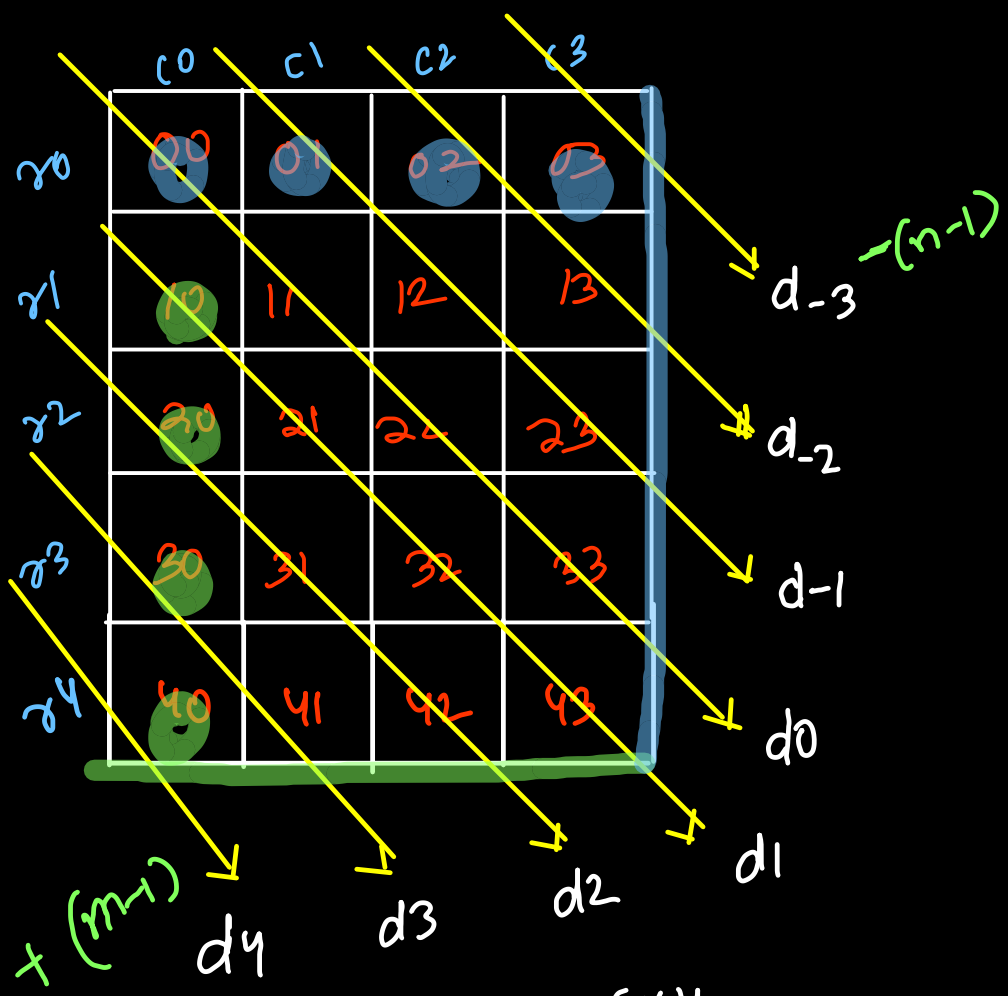
90° anticlockwise = 270° clockwise

270° anticlockwise = 90° clockwise

180° anticlockwise = 180° clockwise

	c0	c1	c2	c3
r0	33	32	31	30
r1	23	22	21	20
r2	13	12	11	10
r3	03	02	01	00

Diagonal Traversal



(1) top left to bottom right:

★ (row - col) same for all indices of a given diagonal

★ outer loop → diagonal

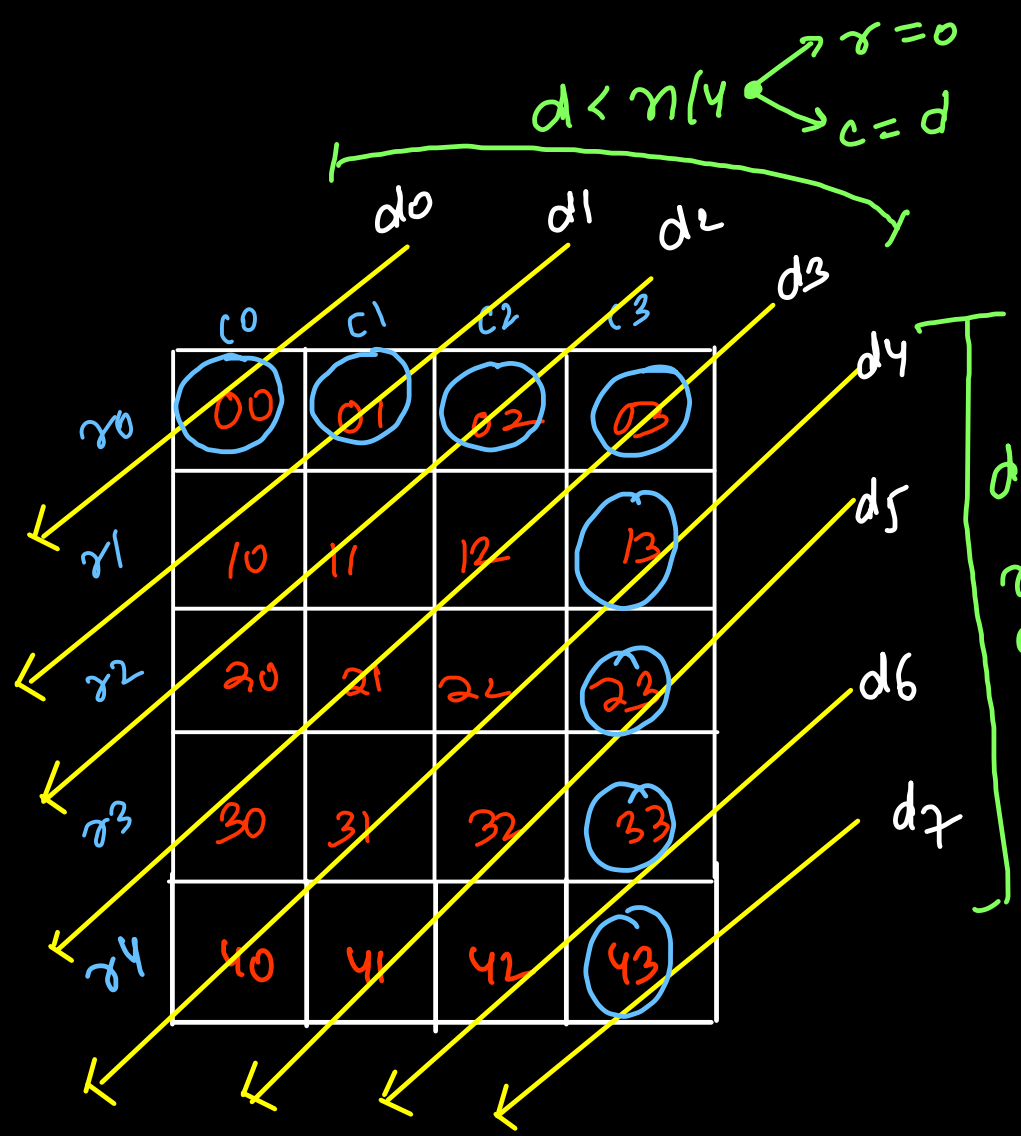
★ inner loop → row, col

Traversal

```
int m = mat.length, n = mat[0].length;
for(int d = 1 - n; d <= m - 1; d++){
    int row = (d <= 0) ? 0 : d;
    int col = (d <= 0) ? -d : 0;

    while(row < m && col < n){
        System.out.print(mat[row][col] + " ");
        row++; col++;
    }
}

return false;
```

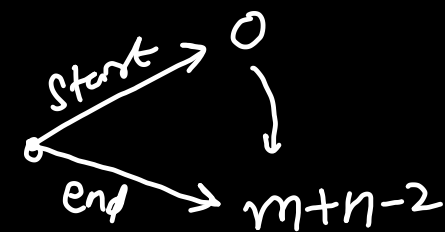
Time ⇒ $O(n^2)$

Space ⇒ $O(1)$



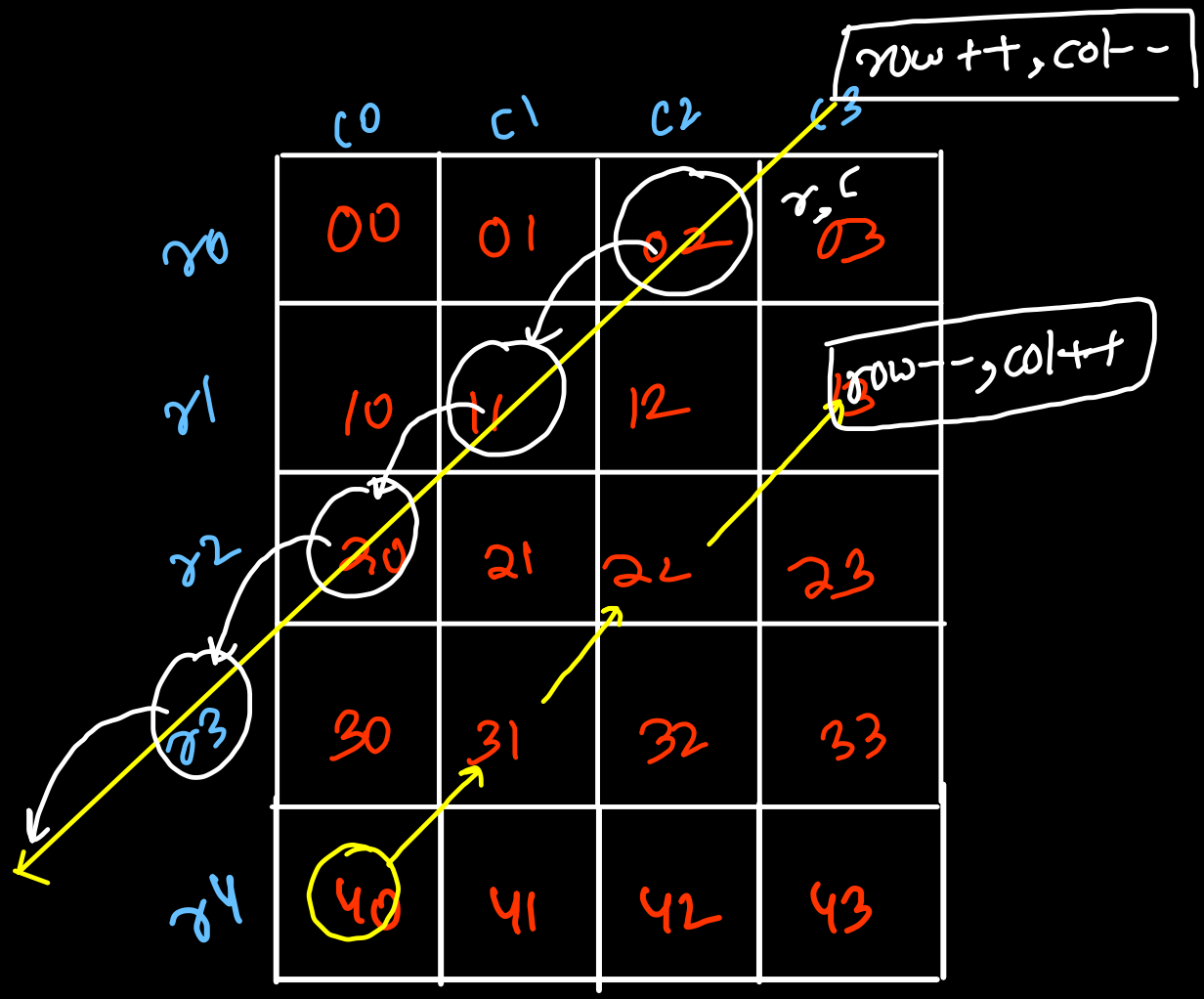
② top right to bottom left

★ (row + col) values for a given diagonal is same



outer loop → diagonal

inner loop → row, col



top right to bottom left

out of matrix

col == -1
left wall

row == m
bottom wall

col >= 0 && row < m

bottom left to top right

row == -1
top wall

col == n
right wall

row >= 0 && col < n

Top Right to Bottom Left

```java
int m = mat.length, n = mat[0].length;

for(int d = 0; d <= m + n - 2; d++){
    int row = (d < n) ? 0 : d - n + 1;
    int col = (d < n) ? d : n - 1;

    while(row < m && col >= 0){
        System.out.print(mat[row][col] + " ");
        row++; col--;
    }
}
```

Time ⇒ O(m*n)

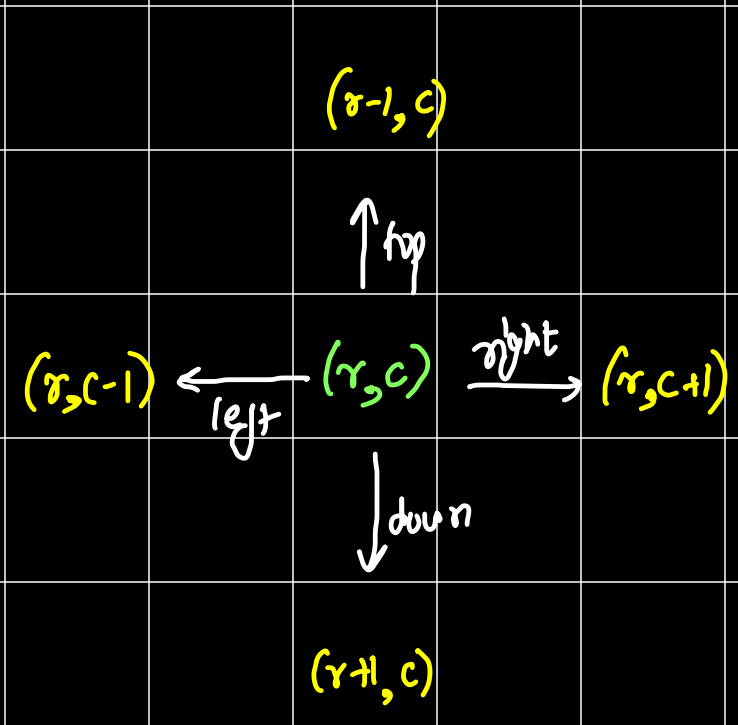
rows cols

quadratic

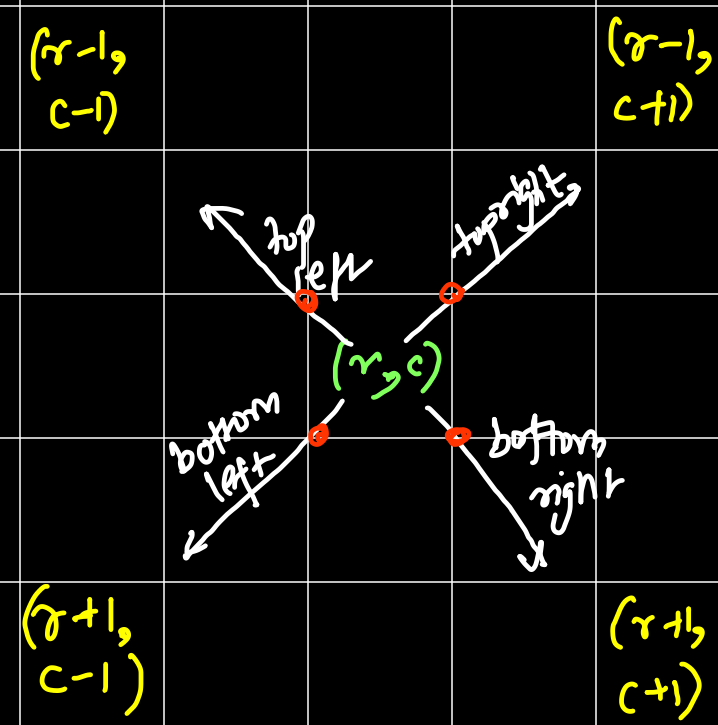
Space ⇒ O(1)

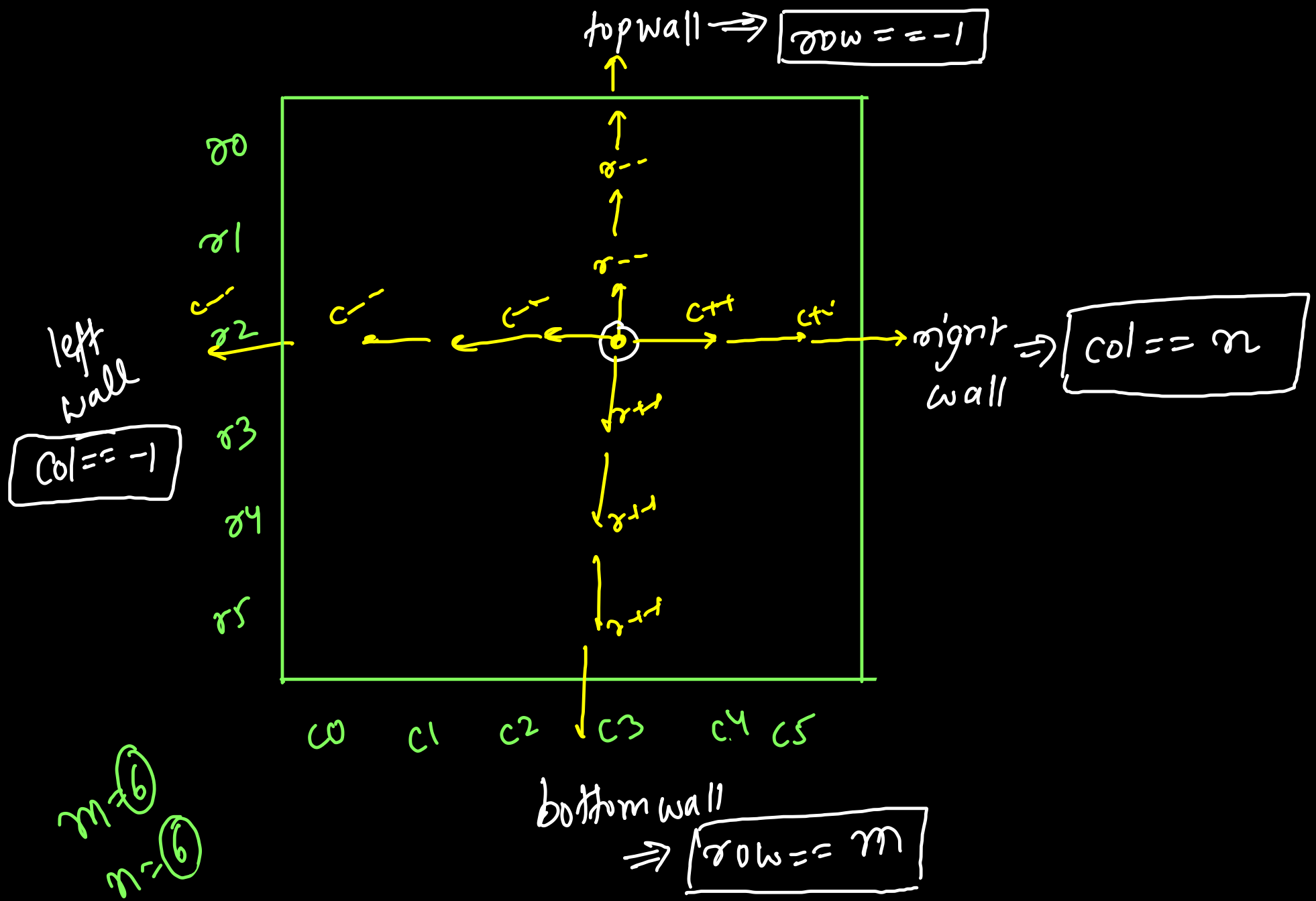
inplace

boundary neighbours

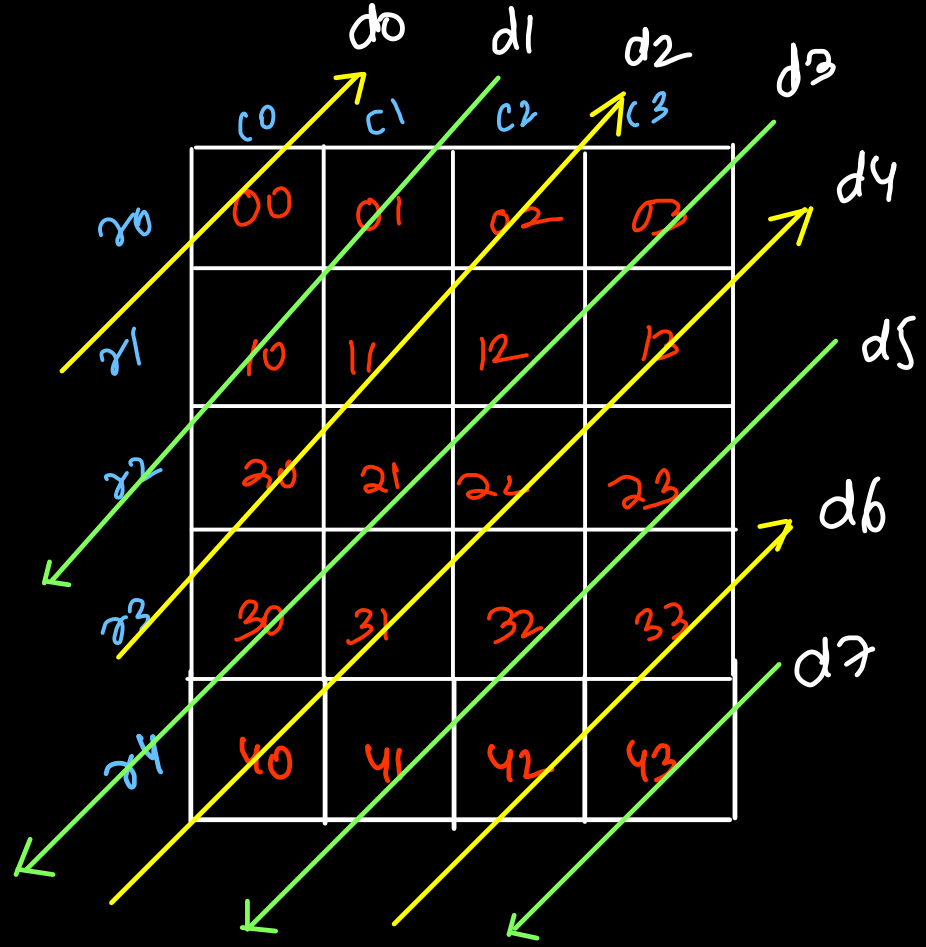


corner adjacent




topwall ⇒ row == -1
r0
r1
r2
r3
r4
r5
r--
c--
r++
c++
left wall
col == -1
right wall ⇒ col == n
c0 c1 c2 c3 c4 c5
bottom wall ⇒ row == m
m = 6
n = 6

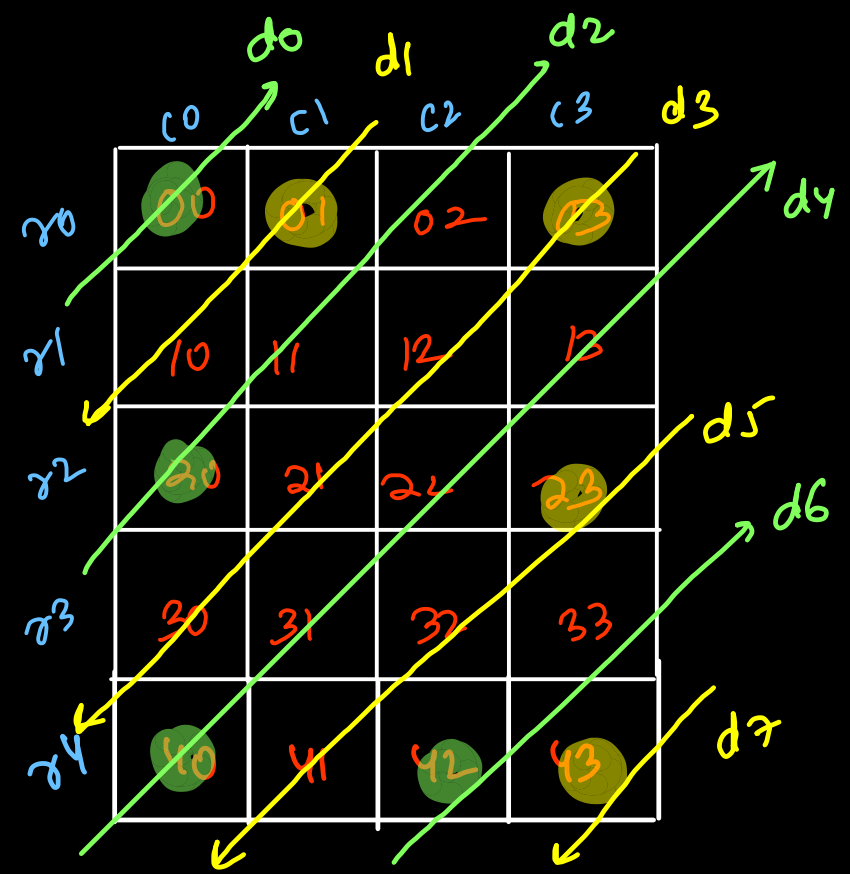
LC 498) ZigZag Traversal



d → odd : top right to bottom left

d → even : bottom left to top right

```java
public int[] findDiagonalOrder(int[][] mat) {
    int m = mat.length, n = mat[0].length;
    int[] res = new int[m * n];
    int idx = 0;

    for(int d = 0; d <= m + n - 2; d++){
        if(d % 2 == 1){
            // top right to bottom left
            int row = (d < n) ? 0 : d - n + 1;
            int col = (d < n) ? d : n - 1;

            while(row < m && col >= 0){
                res[idx++] = mat[row][col];
                row++; col--;
            }
        } else {
            // bottom left to top right
            int row = (d < m) ? d : m - 1;
            int col = (d < m) ? 0 : d - m + 1;

            while(row >= 0 && col < n){
                res[idx++] = mat[row][col];
                row--; col++;
            }
        }
    }

    return res;
```

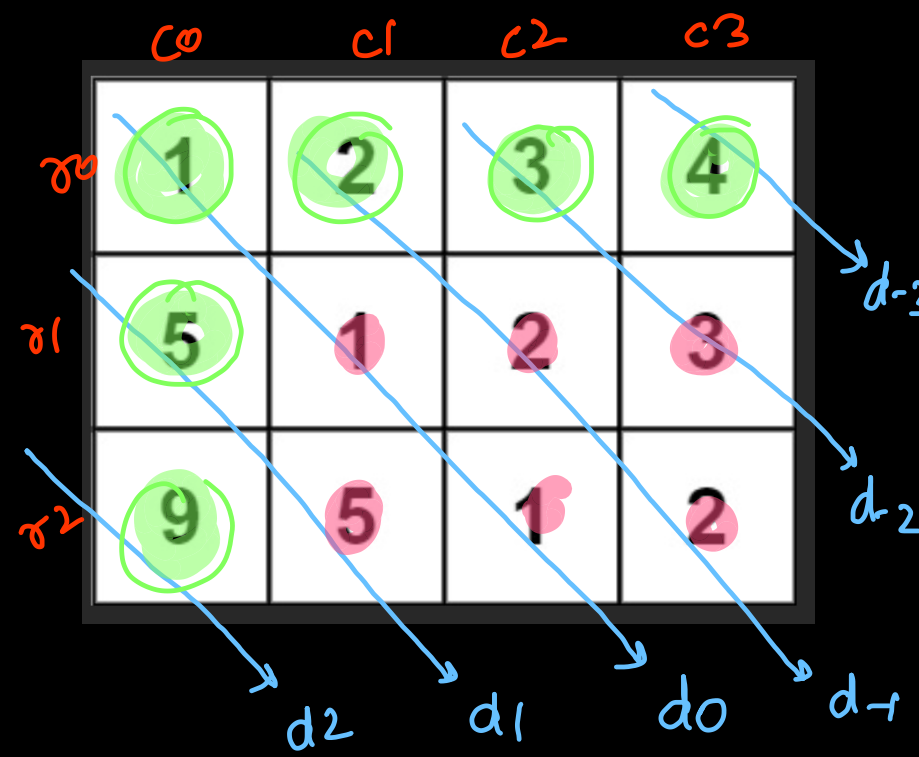


Time ⇒ O(m×n)
quadratic

Space ⇒ O(1)
inplace

LC 766) Toeplitz Matrix

```java
public boolean isToeplitzMatrix(int[][] mat) {
    int m = mat.length, n = mat[0].length;
    for(int d = 1 - n; d <= m - 1; d++){
        int row = (d <= 0) ? 0 : d;
        int col = (d <= 0) ? -d : 0;
        int val = mat[row][col];

        while(row < m && col < n){
            if(mat[row][col] != val) return false;
            row++; col++;
        }
    }

    return true;
}
```



1380. Lucky Numbers in a Matrix

Hint

Easy 1.5K 75

Companies

Given an `m x n` matrix of **distinct** numbers, return *all* ***lucky numbers*** *in the matrix in* ***any*** *order*.

A **lucky number** is an element of the matrix such that it is the minimum element in its row and maximum in its column.

There can be no more than 1 saddle point given that all nos are unique!

eg 1

3	7	8
9	11	13
15	16	17

1 saddle point

eg 2

1	5	8
2	1	7
4	3	1

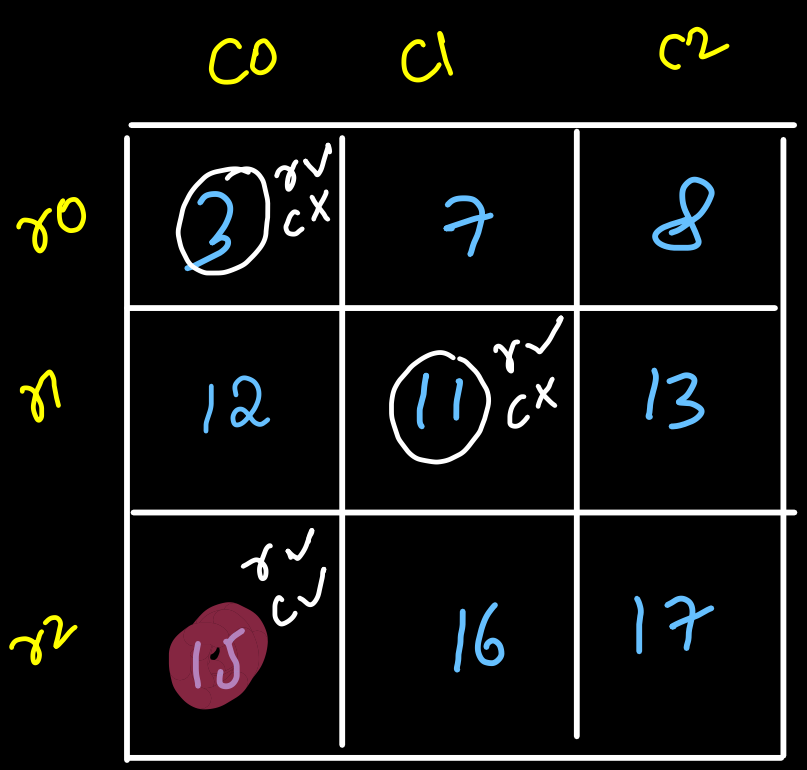
no saddle point

- brute force :→
  - ⇒ check every cell is lucky ⇒ $O(m+n)$

for all cells

⇒ $m \times n \times (m+n)$

⇒ N^3 cubic



r0 → minCol = 0, maxRow = 0 / 2

r1 → minCol = 0 / 1, maxRow = 0 / 1 / 2

r2 → minCol = 0, maxRow = 0 / 1 / 2

```java
public List<Integer> luckyNumbers (int[][] mat) {
    for(int row = 0; row < mat.length; row++){
        int min = 0; // column index
        for(int col = 0; col < mat[0].length; col++){
            if(mat[row][col] < mat[row][min]){
                min = col;
            }
        }

        int max = 0;
        for(int r = 0; r < mat.length; r++){
            if(mat[r][min] > max){
                max = mat[r][min];
            }
        }

        if(max == mat[row][min]) {
            List<Integer> res = new ArrayList<>();
            res.add(max);
            return res;
        }
    }

    return new ArrayList<>();
}
```

min value's col in the current row

max in value in min col

if min == max

Time: quadratic O(n²)

Space: O(1)

→ no saddle point (lucky no)

	C0	C1	C2
r0	8	7	3
r1	12	11	13
r2	15	16	17

r0

min (Col) = 0 ✗ 1 ✗ 2

max = 8 ✗ 7 ✗ 3

17

r1

minCol = 0 ✗ 1

max = 12 ✗ 11 ✗ 16

r2

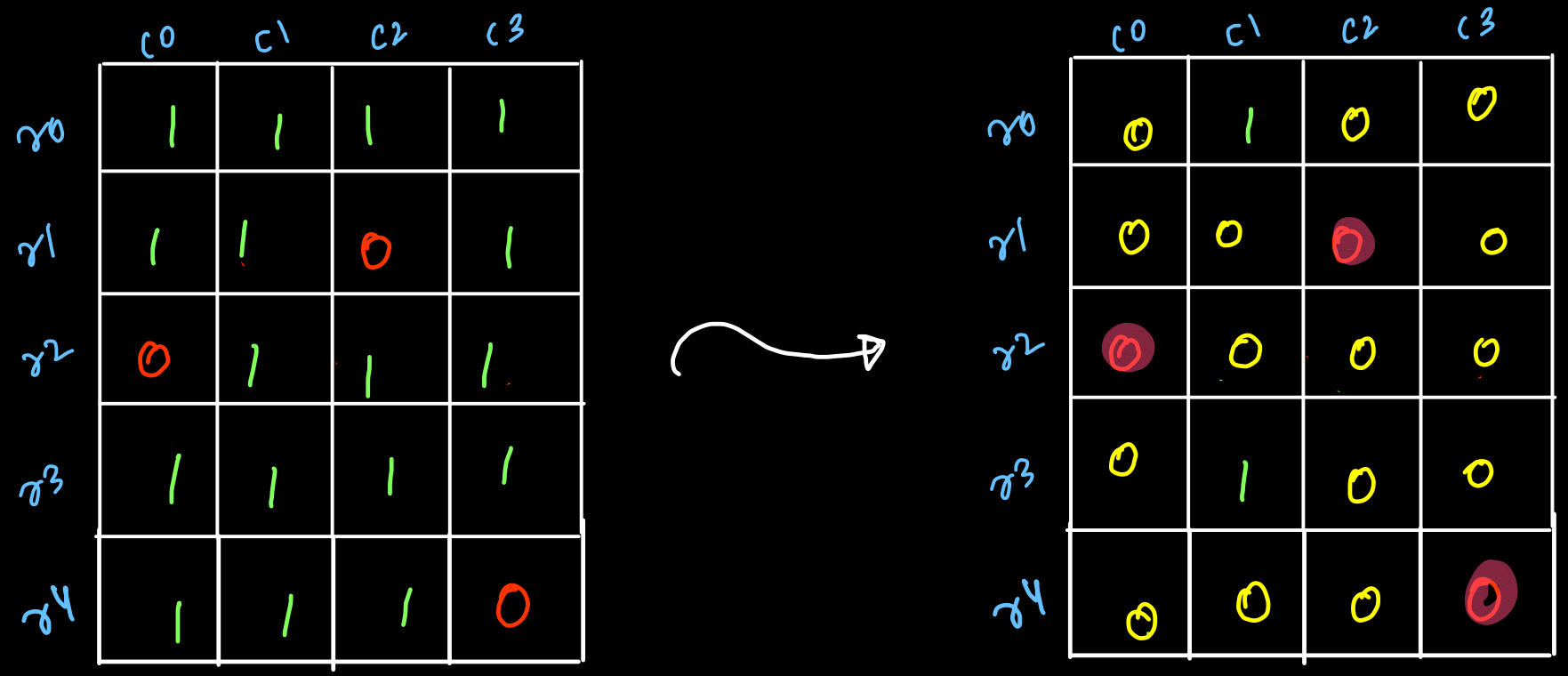
minCol = 0

max = 15 ✗ 16 ✗ 17

{ Google Interview } # Set Matrix Zeroes LC 73

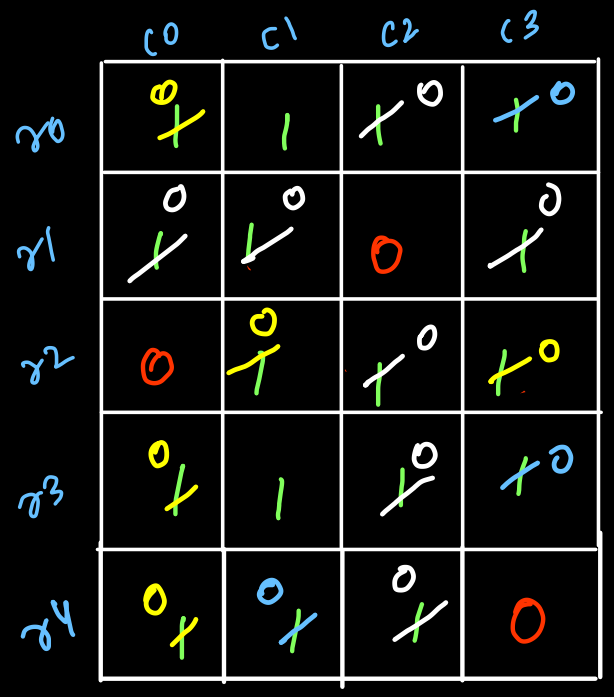
boolean → true(1) / false(0)

int → 0 / non zero (-2^{31} to $2^{31}-1$)

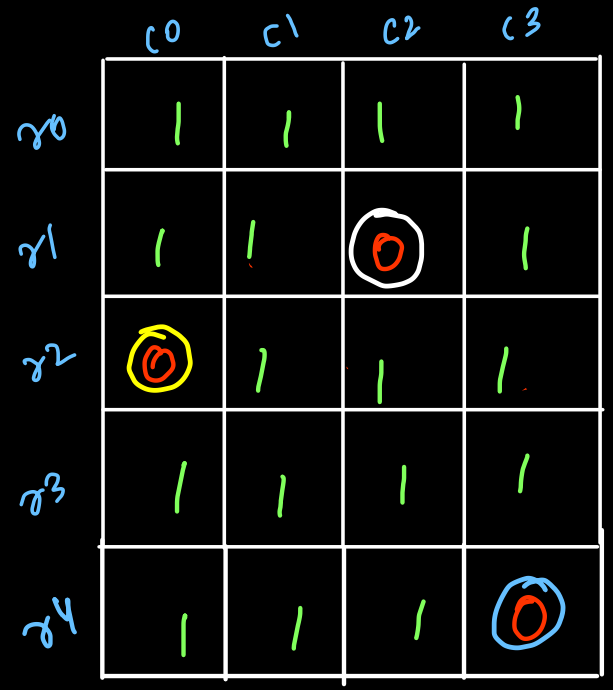


approach1) extra matrix

bruteforce



updates (original)



queries (copy)

```java
public int[][] deepCopy(int[][] mat){
    int m = mat.length, n = mat[0].length;
    int[][] copy = new int[m][n];
    for(int row = 0; row < m; row++){
        for(int col = 0; col < n; col++){
            copy[row][col] = mat[row][col];
        }
    }
    return copy;
}

public void setRowZero(int[][] mat, int row){
    for(int col = 0; col < mat[0].length; col++){
        mat[row][col] = 0;
    }
}

public void setColZero(int[][] mat, int col){
    for(int row = 0; row < mat.length; row++){
        mat[row][col] = 0;
    }
}
```

```java
public void setZeroes(int[][] mat) {
    int[][] copy = deepCopy(mat);

    for(int row = 0; row < copy.length; row++){
        for(int col = 0; col < copy[0].length; col++){
            if(copy[row][col] == 0){
                setRowZero(mat, row);
                setColZero(mat, col);
            }
        }
    }
}
```

extra space

m

n

m+n

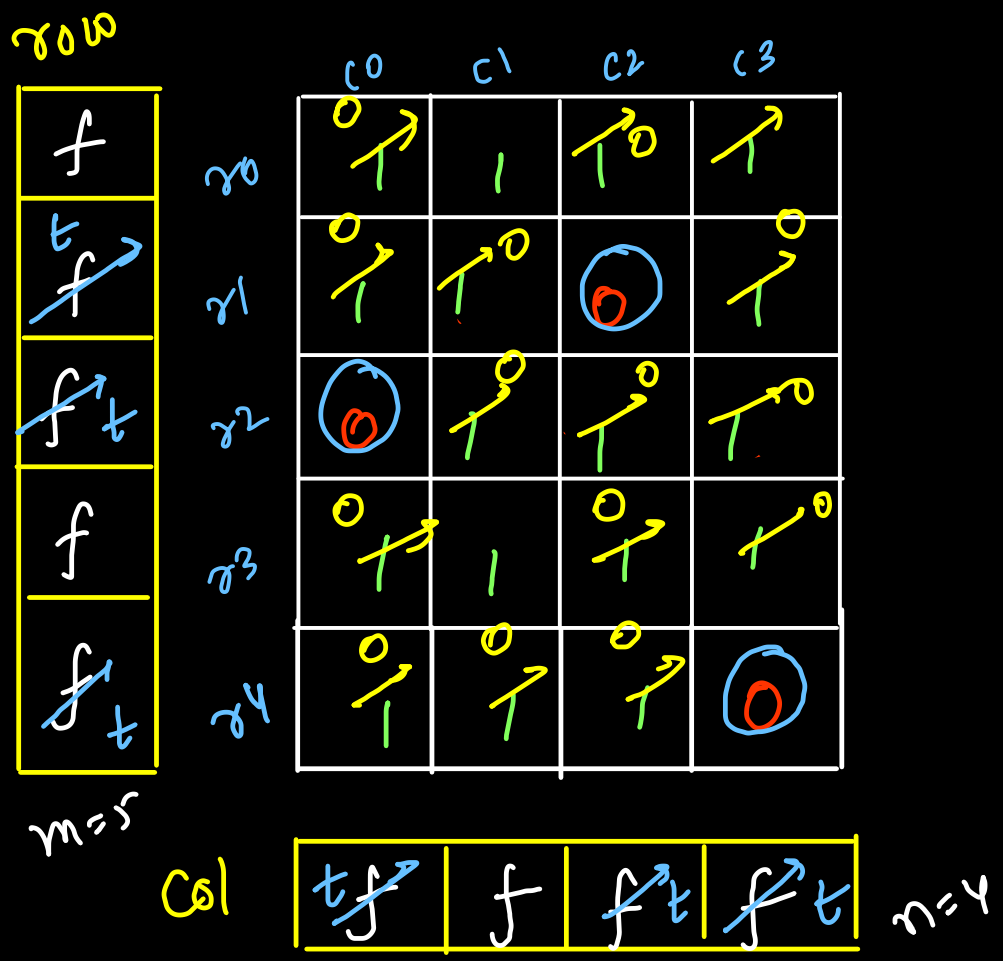
Time ⇒ O(m x n x (m+n)) = cubic

Space ⇒ O(m x n) quadratic

~~in place~~

2nd approach (lazy)

- (1) queries ⇒ zeros (store)
- then
- (2) updates (set zeros)



queries ⇒ O(m×n)

+

updates ⇒ O(m×n)

Time = O(m×n) quadratic

Space = O(m+n) linear
(extra space)

~~in place~~

2nd

```java
public void setZeroes(int[][] mat) {
    boolean[] rows = new boolean[mat.length];
    boolean[] cols = new boolean[mat[0].length];

    // Queries -> Store Zeroes
    for(int row = 0; row < mat.length; row++){
        for(int col = 0; col < mat[0].length; col++){
            if(mat[row][col] == 0){
                rows[row] = cols[col] = true;
            }
        }
    }

    // Updates -> Set Zeros
    for(int row = 0; row < mat.length; row++){
        for(int col = 0; col < mat[0].length; col++){
            if(rows[row] == true || cols[col] == true){
                mat[row][col] = 0;
            }
        }
    }
}
```

extra space

1st → 2nd → 3rd

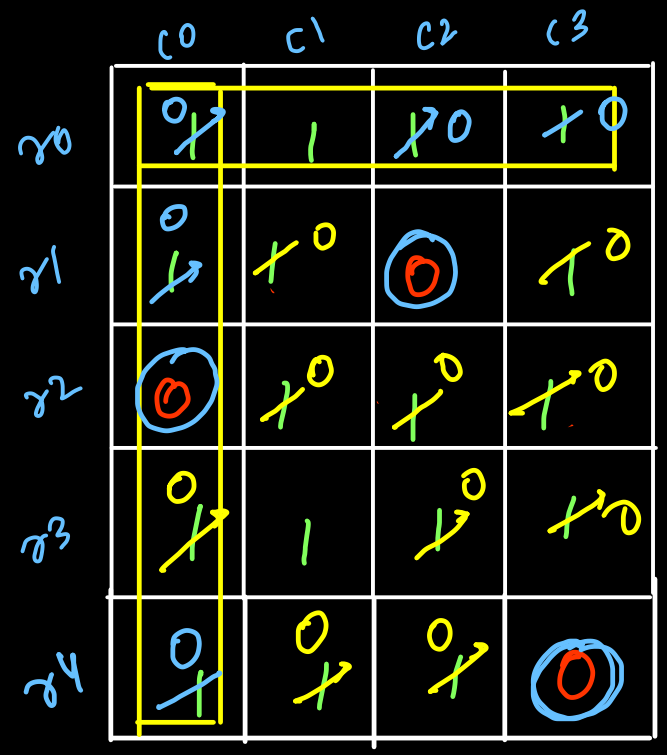
$O(N^3)$ time ⇒ $O(N^2)$ time ⇒ $O(N^2)$ time

$O(N^2)$ Space ⇒ $O(N)$ Space ⇒ $O(1)$ space (in place)

(3)rd approach (most optimized)

Time ⇒ $O(n^2)$

Space ⇒ $O(1)$ inplace



boolean firstrow = false;

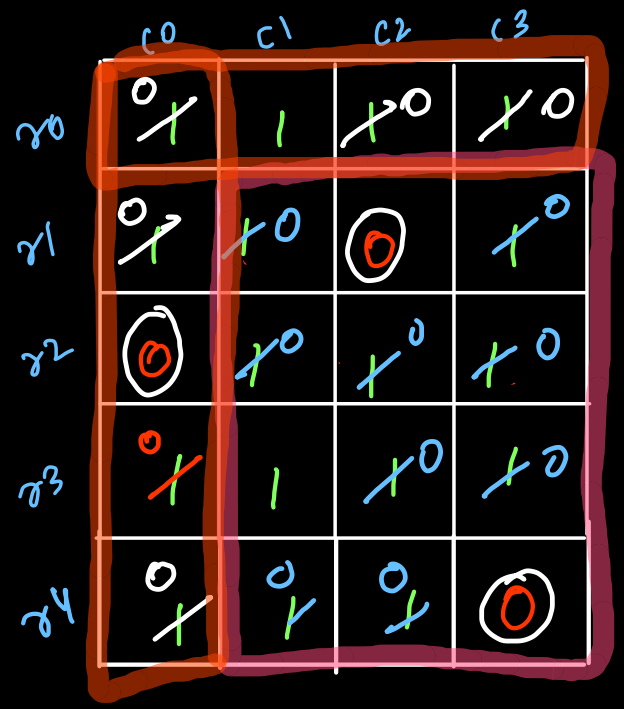
boolean firstcol = ~~false~~ true;

(1) queries → store zeros

(2) update → set zeros

time
O(m×n)
Quadratic

Space
O(1)
Constant
(inplace)



r0 = false .
c0 = ~~false~~ true

```java
public void setZeroes(int[][] mat) {
    boolean r0Zero = false, c0Zero = false;

    // Queries -> Store Zeroes
    for(int row = 0; row < mat.length; row++){
        for(int col = 0; col < mat[0].length; col++){
            if(mat[row][col] == 0){
                mat[0][col] = mat[row][0] = 0;
                if(row == 0) r0Zero = true;
                if(col == 0) c0Zero = true;
            }
        }
    }

    // Updates -> Set Zeros
    for(int row = 1; row < mat.length; row++){
        for(int col = 1; col < mat[0].length; col++){
            if(mat[row][0] == 0 || mat[0][col] == 0){
                mat[row][col] = 0;
            }
        }
    }

    // first Row (r0)
    if(r0Zero == true){
        for(int col = 0; col < mat[0].length; col++){
            mat[0][col] = 0;
        }
    }

    // first Col (c0)
    if(c0Zero == true){
        for(int row = 0; row < mat.length; row++){
            mat[row][0] = 0;
        }
    }
}
```

O(m×n)

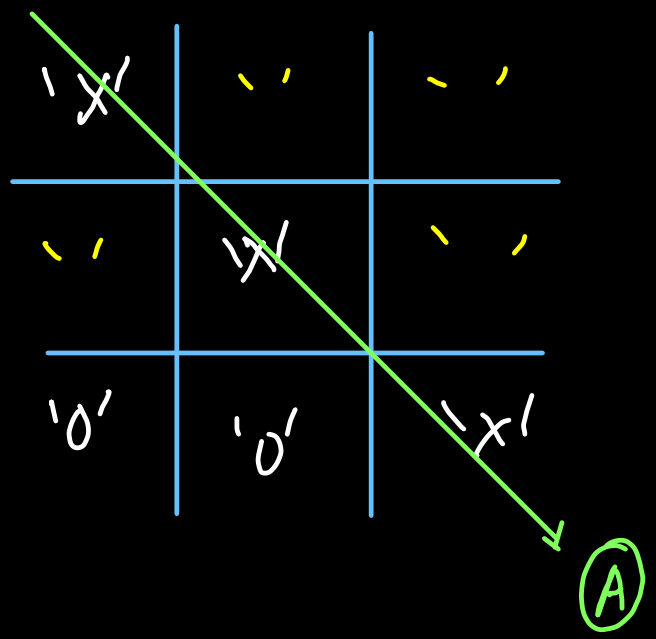
* corner case

LC 1275) Tic Tac Toe Game

eg1

```
moves = [[0,0],[2,0],[1,1],[2,1],[2,2]]
```

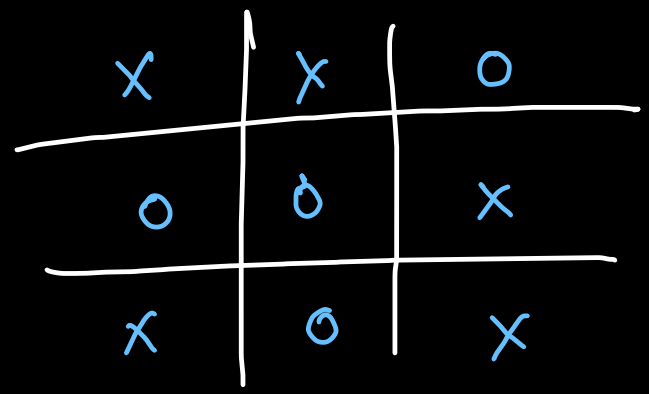
move = 'X' 'O' 'X' 'O' 'X'




```
moves = [[0,0],[1,1],[2,0],[1,0],[1,2],[2,1],[0,1],[0,2],[2,2]]
```

"Pending" "Draw"

X O X O X O X O X



```java
public boolean isWinner(char[][] board, int row, int col, char ch){
    // entire row
    if(board[row][0] == ch && board[row][1] == ch && board[row][2] == ch)
        return true;

    // entire column
    if(board[0][col] == ch && board[1][col] == ch && board[2][col] == ch)
        return true;

    // left diagonals
    if(board[0][0] == ch && board[1][1] == ch && board[2][2] == ch)
        return true;

    // right diagonals
    if(board[0][2] == ch && board[1][1] == ch && board[2][0] == ch)
        return true;

    return false;
}
```

↳ Constant O(1)

```java
public String tictactoe(int[][] moves) {
    char ch = 'X';
    char[][] board = new char[3][3];

    for(int[] move: moves){
        int row = move[0];
        int col = move[1];
        board[row][col] = ch;

        if(isWinner(board, row, col, ch) == true){
            if(ch == 'X') return "A";
            else return "B";
        }

        if(ch == 'X') ch = 'O';
        else ch = 'X';
    }

    if(moves.length == 9) return "Draw";
    else return "Pending";
}
```

Worst case ⇒ 9 ⇒ O(1)

LC 36) Valid Sudoku

Determine if a `9 x 9` Sudoku board is valid. Only the filled cells need to be validated **according to the following rules**:

Each row must contain the digits `1-9` without repetition.

Each column must contain the digits `1-9` without repetition.

Each of the nine `3 x 3` sub-boxes of the grid must contain the digits `1-9` without repetition.

Note:

- A Sudoku board (partially filled) could be valid but is not necessarily solvable.
- Only the filled cells need to be validated according to the mentioned rules.

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

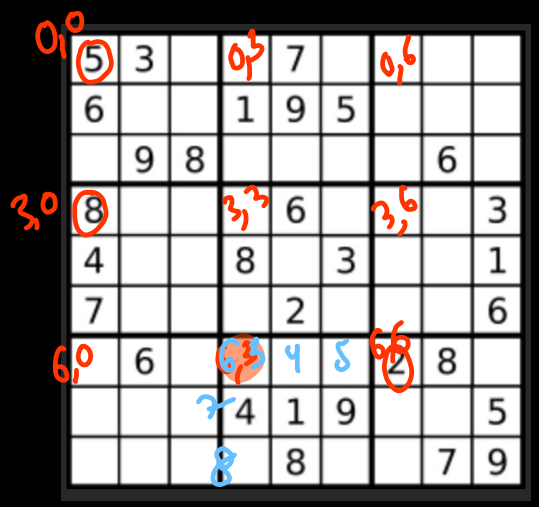
```java
public boolean isRowValid(char[][] board, int row){
    int[] freq = new int[10];
    for(int col = 0; col < 9; col++){
        int val = board[row][col] - '0';
        freq[val]++;
        if(freq[val] > 1) return false;
    }
    return true;
}

public boolean isColValid(char[][] board, int col){
    int[] freq = new int[10];
    for(int row = 0; row < 9; row++){
        int val = board[row][col] - '0';
        freq[val]++;
        if(freq[val] > 1) return false;
    }
    return true;
}
```

```java
public boolean isSubGridValid(char[][] board){

}

public boolean isValidSudoku(char[][] board) {
    for(int row = 0; row < 9; row++){
        if(isRowValid(board, row) == false) return false;
    }

    for(int col = 0; col < 9; col++){
        if(isColValid(board, col) == false) return false;
    }


}
```

(0,0) (3,0) (6,0)

(0,3) (3,3) (6,3)

(0,6) (3,6) (6,6)




```java
public boolean isRowValid(char[][] board, int row){
    int[] freq = new int[10];
    for(int col = 0; col < 9; col++){
        if(board[row][col] == '.') continue;
        int val = board[row][col] - '0';
        freq[val]++;
        if(freq[val] > 1) return false;
    }
    return true;
}

public boolean isColValid(char[][] board, int col){
    int[] freq = new int[10];
    for(int row = 0; row < 9; row++){
        if(board[row][col] == '.') continue;
        int val = board[row][col] - '0';
        freq[val]++;
        if(freq[val] > 1) return false;
    }
    return true;
}
```

O(9)

O(9)

```java
public boolean isSubGridValid(char[][] board, int row, int col){
    int[] freq = new int[10];
    for(int r = row; r < row + 3; r++){
        for(int c = col; c < col + 3; c++){
            if(board[r][c] == '.') continue;
            int val = board[r][c] - '0';
            freq[val]++;
            if(freq[val] > 1) return false;
        }
    }
    return true;
}

public boolean isValidSudoku(char[][] board) {
    for(int row = 0; row < 9; row++){
        if(isRowValid(board, row) == false) return false;
    }

    for(int col = 0; col < 9; col++){
        if(isColValid(board, col) == false) return false;
    }

    for(int row = 0; row < 9; row += 3){
        for(int col = 0; col < 9; col += 3){
            if(isSubGridValid(board, row, col) == false)
                return false;
        }
    }

    return true;
}
```

O(9)

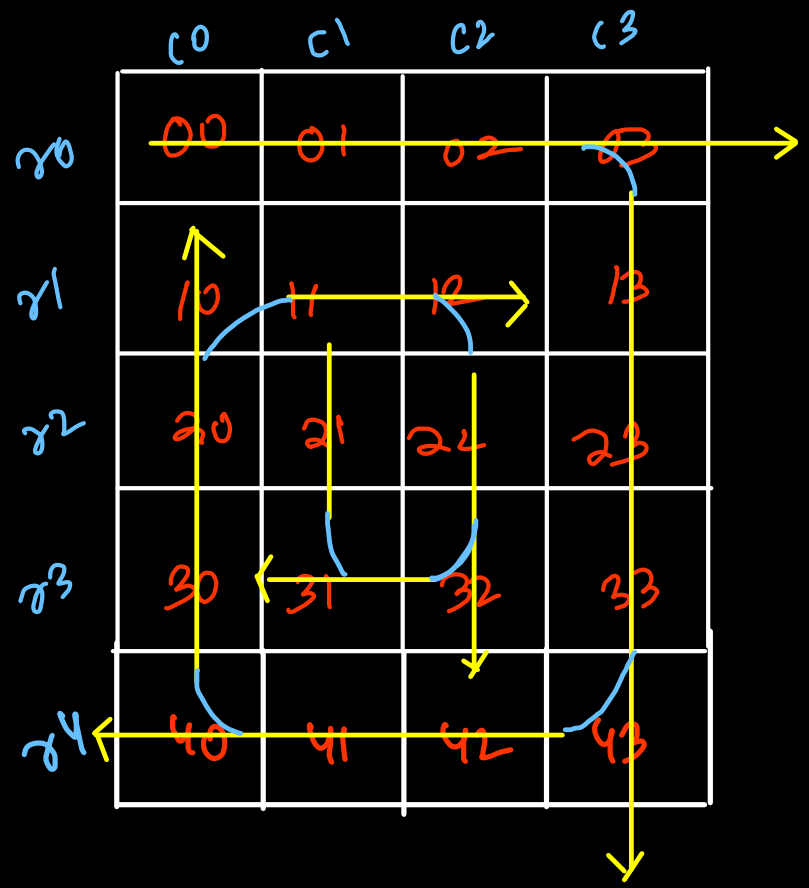
O(3x9x9)

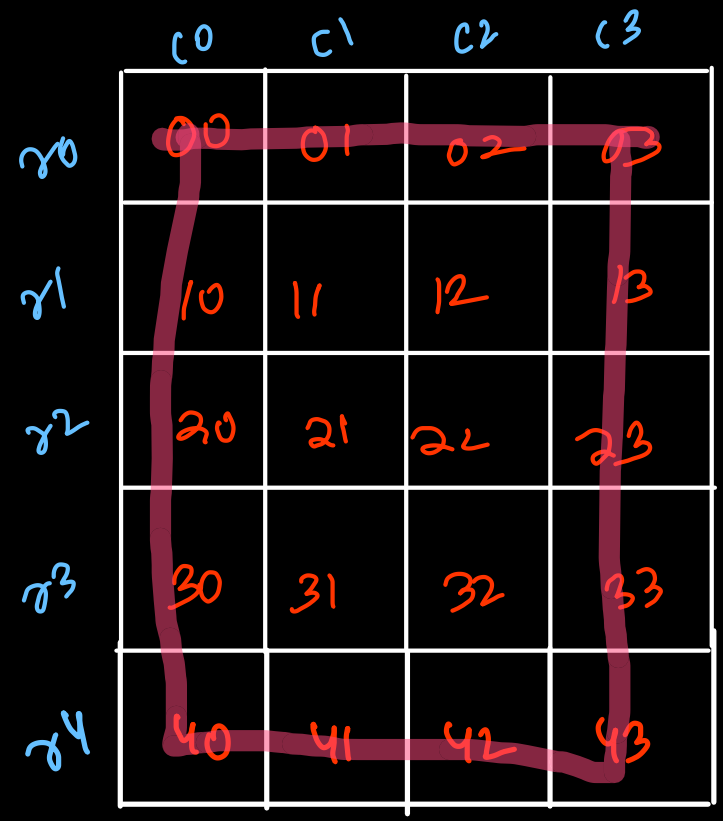
9x9

9x9

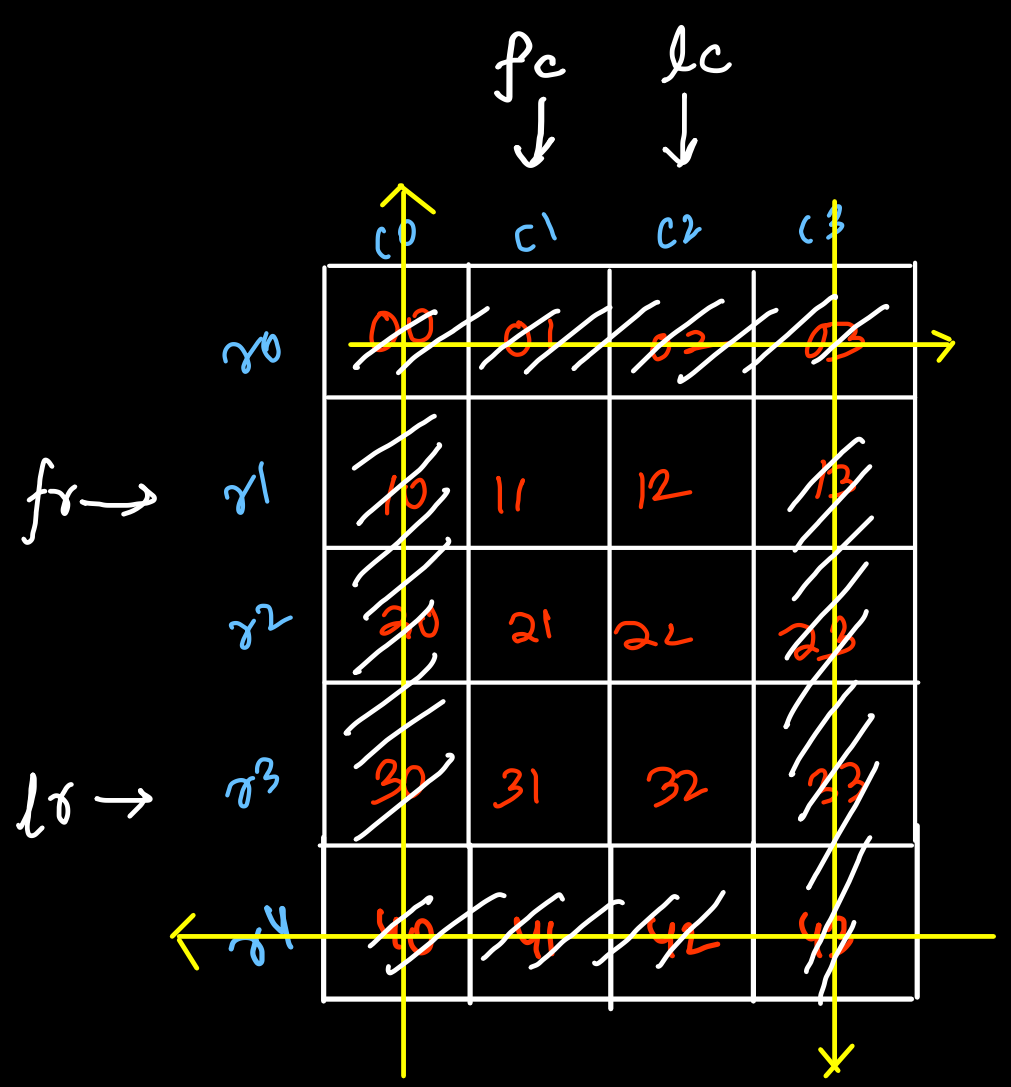
9x9

LC 54) Spiral matrix





boundary traversal




```
fr=0, fc=0, lr=m-1, lc=n-1;

// top wall (left to right)
for(int c=fc; c<=lc; c++)
    Syso(mat[fr][c]);
fr++;

// right wall (top to bottom)
for(int r=fr; r<=lr; r++)
    Syso(mat[r][lc]);
lc--;

// bottom wall (right to left)
for(int c=lc; c>=fc; c--)
    Syso(mat[lr][c]);
lr--;

// left wall (bottom to top)
for(int r=lr; r>=fr; r--)
    Syso(mat[r][fc]);
fc++
```

```java
public List<Integer> spiralOrder(int[][] mat) {
    List<Integer> spiral = new ArrayList<>();
    int fr = 0, fc = 0, lr = mat.length - 1, lc = mat[0].length - 1;

    while(true){ → allshells
        // top wall (left to right)
        for(int c = fc; c <= lc; c++){
            spiral.add(mat[fr][c]);
        }
        fr++; → ignore top wall
        if(fr > lr) break;
        // right wall (top to bottom)
        for(int r = fr; r <= lr; r++){
            spiral.add(mat[r][lc]);
        }
        lc--; → ignore
        if(fc > lc) break;
        // bottom wall (right to left)
        for(int c = lc; c >= fc; c--){
            spiral.add(mat[lr][c]);
        }
        lr--; → ignore
        if(fr > lr) break;
        // left wall (bottom to top)
        for(int r = lr; r >= fr; r--){
            spiral.add(mat[r][fc]);
        }
        fc++; → ignore
        if(fc > lc) break;
    }

    return spiral;
}
```

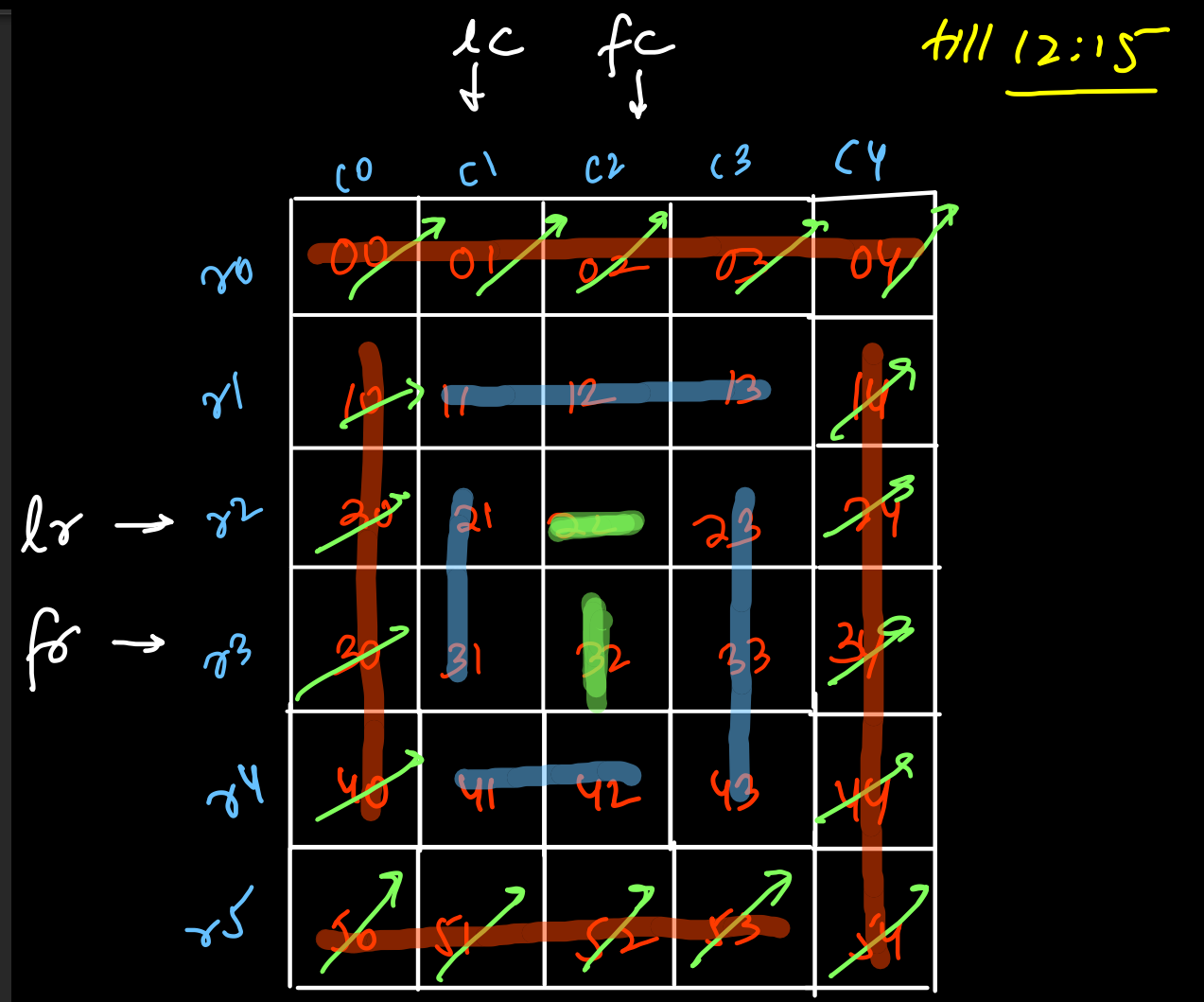
ignore shell & all walls

no row remaining

no column remaining

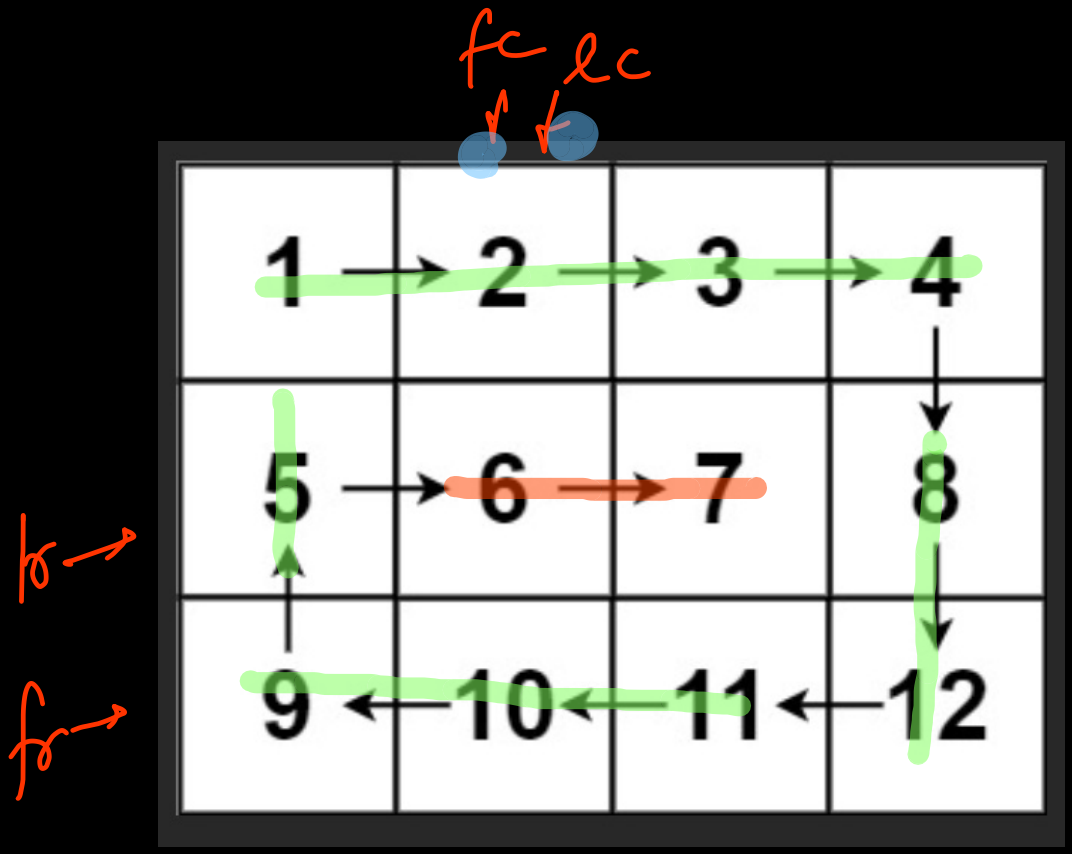
no row rem

no col rem



Time : O(mxn) quadratic

Space : O(1) inplace

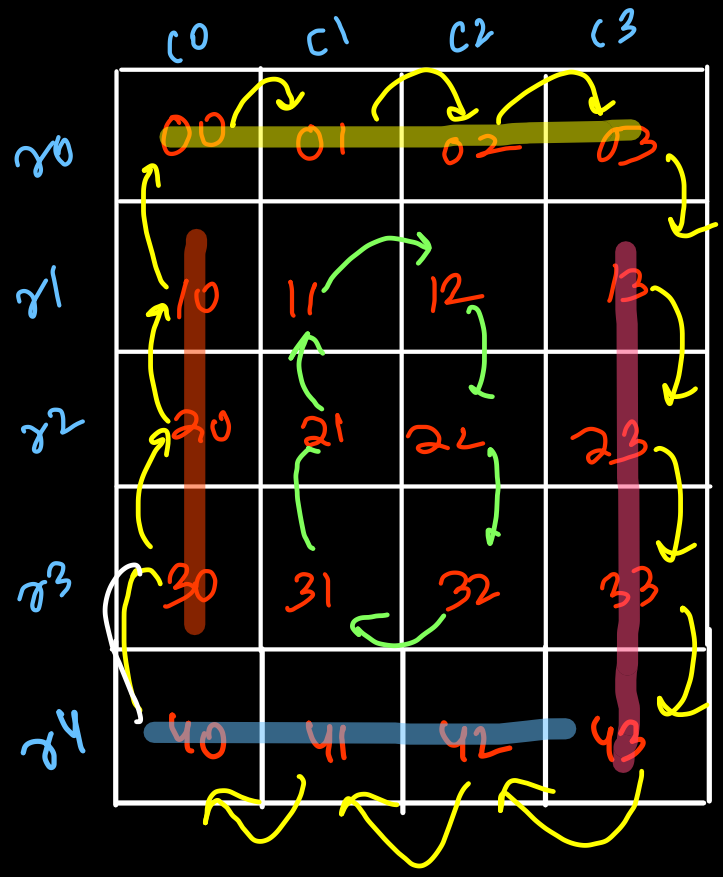



```java
public List<Integer> spiralOrder(int[][] mat) {
    List<Integer> spiral = new ArrayList<>();
    int fr = 0, fc = 0, lr = mat.length - 1, lc = mat[0].length - 1;

    while(true){
        // top wall (left to right)
        for(int c = fc; c <= lc; c++){
            spiral.add(mat[fr][c]);
        }
        fr++;
        if(fr > lr) break;

        // right wall (top to bottom)
        for(int r = fr; r <= lr; r++){
            spiral.add(mat[r][lc]);
        }
        lc--;
        if(fc > lc) break;

        // bottom wall (right to left)
        for(int c = lc; c >= fc; c--){
            spiral.add(mat[lr][c]);
        }
        lr--;
        if(fr > lr) break;

        // left wall (bottom to top)
        for(int r = lr; r >= fr; r--){
            spiral.add(mat[r][fc]);
        }
        fc++;
        if(fc > lc) break;
    }

    return spiral;
}
```

Homework

GFG

Rotate Shells Of matrix



	c0	c1	c2	c3
r0	10	00	01	02
r1	20	21	11	03
r2	30	31	12	13
r3	40	32	22	23
r4	41	42	43	33

Shell by shell

↓

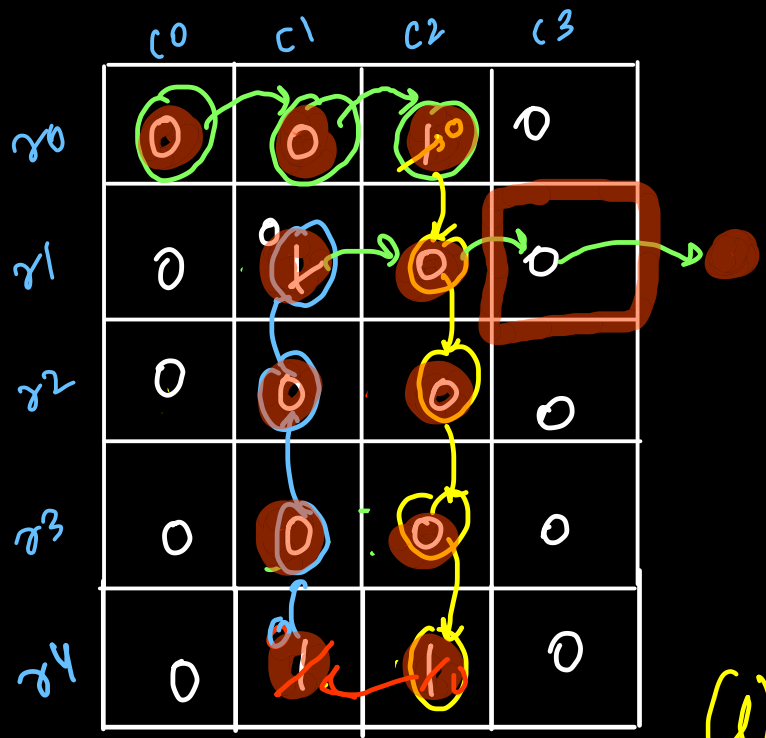
⇒ top wall ⇒ arraylist
↳ right shift

⇒ rightwall ⇒ arraylist
↳ downshift

⇒ bottom wall ⇒ arraylist
↳ left shift

⇒ left wall ⇒ arraylist
↳ top shift

Exit Point of Matrix



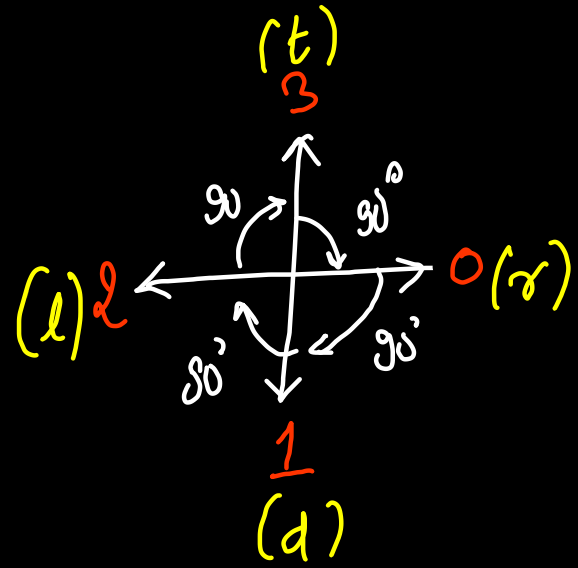
Starting cell ⇒ (0,0)

Starting direction ⇒ left to right

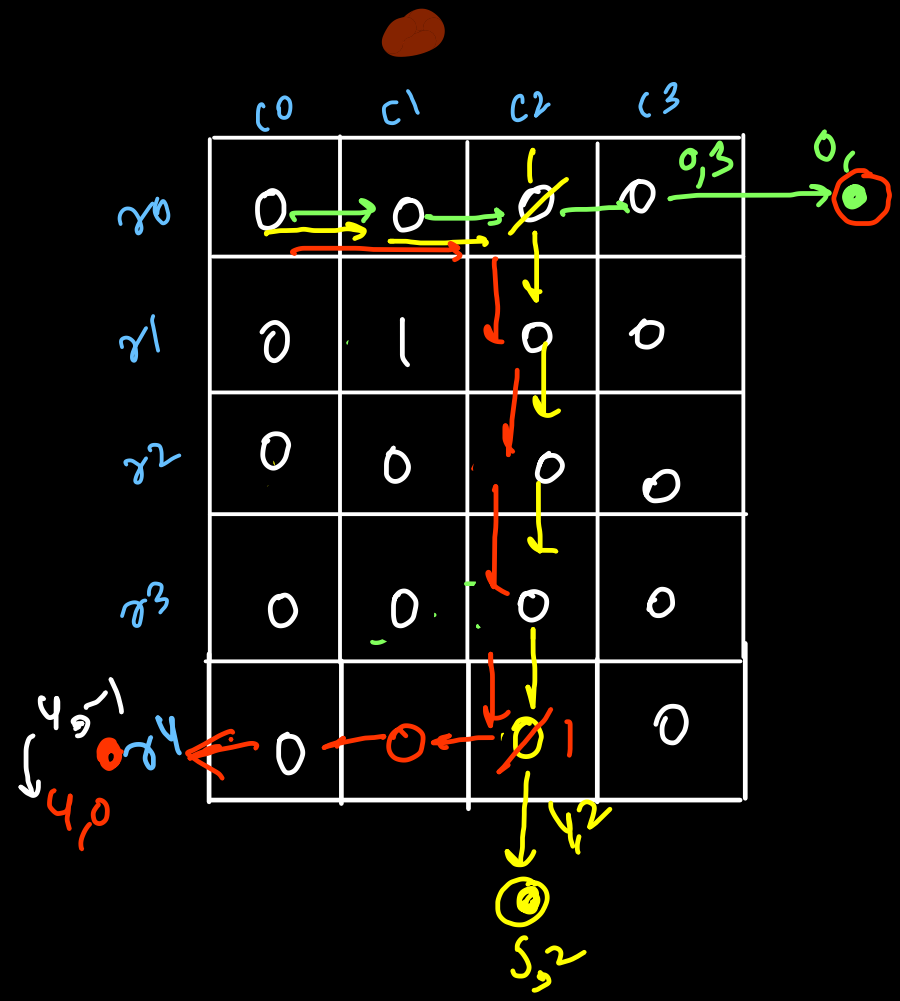
0 ⇒ same direction

1 ⇒ direction ⇒ 90° rightwards

change value to 0.



```java
// direction = 0 -> right, 1 -> down, 2 -> left, 3 -> top
public int[] FindExitPoint(int[][] mat)
{
    int row = 0, col = 0, direction = 0;

    while(row >= 0 && col >= 0 && row < mat.length && col < mat[0].length){
        if(mat[row][col] == 1){
            mat[row][col] = 0; // set 1 to 0
            direction = (direction + 1) % 4; // rotate the direction
        }

        if(direction == 0){
            // right
            col++;
        } else if(direction == 1){
            // down
            row++;
        } else if(direction == 2){
            // left
            col--;
        } else {
            // top
            row--;
        }
    }
```



```java
// direction = 0 -> right, 1 -> down, 2 -> left, 3 -> top
public int[] FindExitPoint(int[][] mat)
{
    int row = 0, col = 0, direction = 0;

    while(row >= 0 && col >= 0 && row < mat.length && col < mat[0].length){
        if(mat[row][col] == 1){
            mat[row][col] = 0; // set 1 to 0
            direction = (direction + 1) % 4; // rotate the direction
        }

        if(direction == 0){
            // right
            col++;
        } else if(direction == 1){
            // down
            row++;
        } else if(direction == 2){
            // left
            col--;
        } else {
            // top
            row--;
        }
    }
```

```java
if(row < 0){
    // top wall
    return new int[]{row + 1, col};
} else if(col < 0){
    // left wall
    return new int[]{row, col + 1};
} else if(row == mat.length){
    // bottom wall
    return new int[]{row - 1, col};
} else {
    // right wall
    return new int[]{row, col - 1};
}
```

Calculate last cell within matrix

Time ↓

O(n x m)

Space ↓

O(1) inplace