Representation of Matrix:-

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

(2×1)

2D array

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

(2×3)

2D array

A matrix is a 2D array represented by rows and columns.

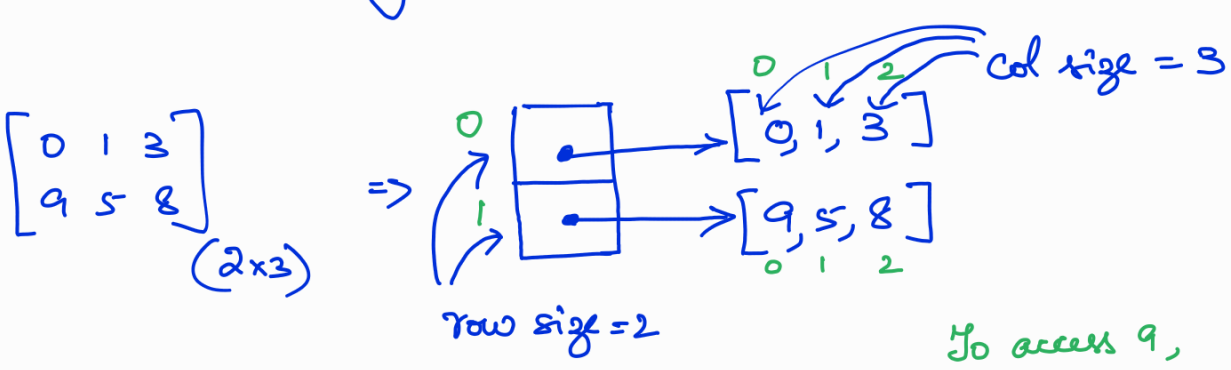
The number of rows is the size of the outer array

The number of colums is the size of each inner array.

The outer array stores references to the inner arrays.

$$\begin{bmatrix} 0 & 1 & 3 \\ 9 & 5 & 8 \end{bmatrix}$$

(2×3)



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

To access 9,

A[1][0]

To access 3,

A[0][2]

row ref, col ref

Operations:-

(1) Addition & subtraction:-

Two matrices should have same dimensions.

addition

1	2	3
4	5	6

2×3

$\pm$

4	5	6
1	3	2

2×3

$=$

5	7	9
5	8	8

2×3

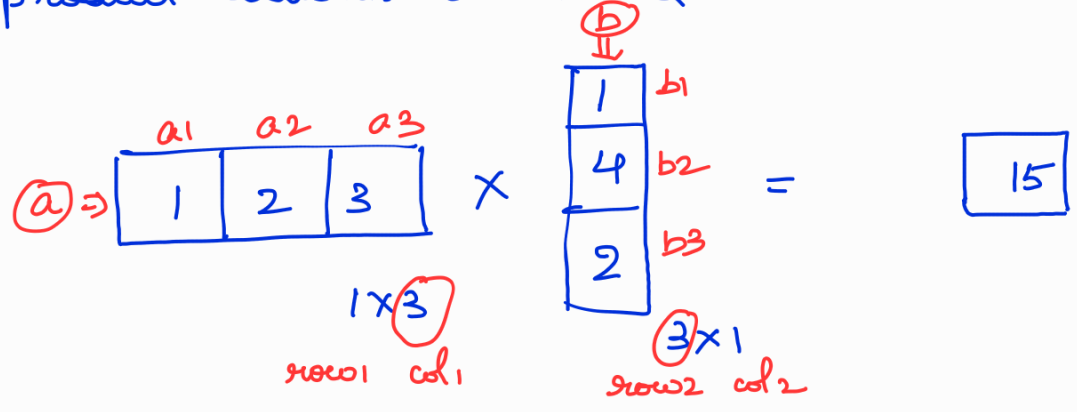
subtraction

-3	-3	-3
3	2	4

2×3

② Multiplication :-

$a \cdot b =$ dot product between a row a & column b is $\Rightarrow a_1b_1 + a_2b_2 + a_3b_3$



if col 1 = row 2 then multiply

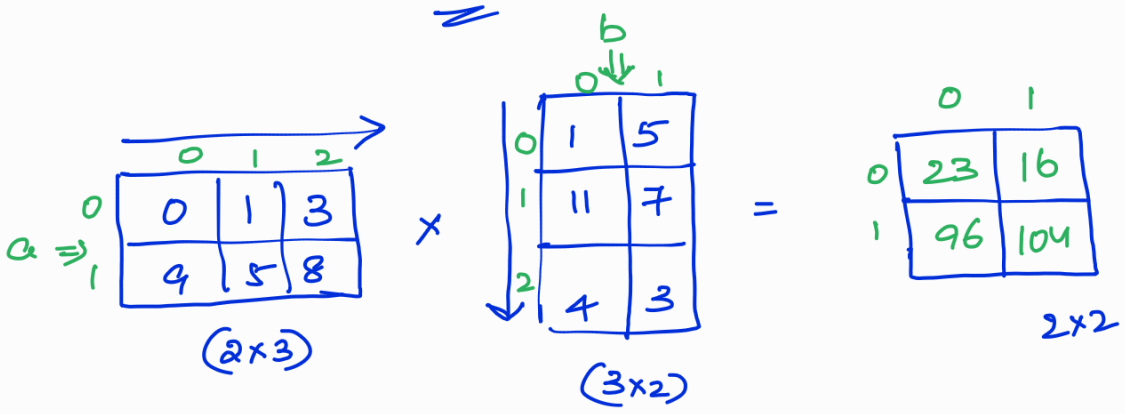
result = row 1 × col 2

dot product = $(1\times1) + (2\times4) + (3\times2)$

$= 1+8+6$

$= 15$

Eg 2 :-



a row 0 | b col 0

dot product (00) $\Rightarrow a_{00}b_{00} + a_{01}b_{10} + a_{02}b_{20}$

a row 0 | b col 1

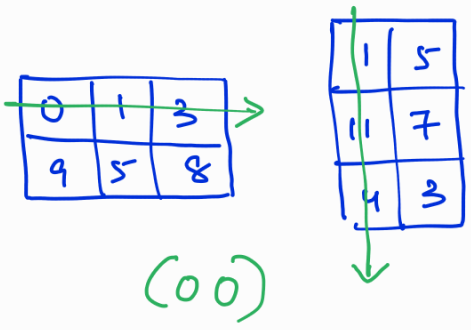
(01) $\Rightarrow a_{00}b_{01} + a_{01}b_{11} + a_{02}b_{21}$

a row 1 | b col 0

(10) $\Rightarrow a_{10}b_{00} + a_{11}b_{10} + a_{12}b_{20}$

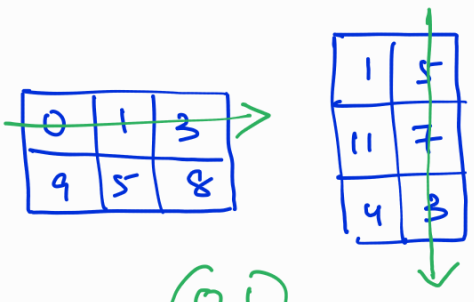
a row 2 | b col 1

(11) $\Rightarrow a_{10}b_{01} + a_{11}b_{11} + a_{12}b_{21}$



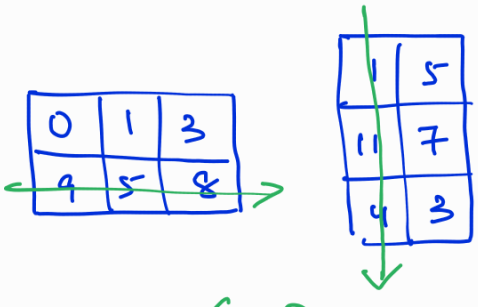
$\Rightarrow (0\times1) + (1\times11) + (3\times4) = 23$

(00)



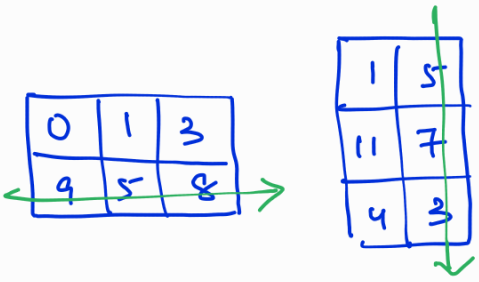
$\Rightarrow (0\times5) + (1\times7) + (3\times3) = 16$

(01)



$\Rightarrow (9\times1) + (5\times11) + (8\times4) = 96$

(10)



$\Rightarrow (9\times5) + (5\times7) + (8\times3) = 104$

(11)

Ans:-

00	01
23	16
10	11
96	104

(2 x 2)

③ Inverse:- For a square matrix A, if there exists another square matrix B such that $AB = BA = I$ where I is identity matrix, B is called inverse of A, denoted as A^{-1}

1	5	2
0	-1	2
0	0	1

A

×

1	5	-12
0	-1	2
0	0	1

$B = A^{-1}$

=

1	0	0
0	1	0
0	0	1

I

A^{-1} calculated as: for $\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$

$$\begin{bmatrix} -[ei-fh] & [di-fg] & -[dh-eg] \\ [bi-ch] & -[ai-cg] & [ah-bg] \\ -[bf-ce] & [af-cd] & -[ae-db] \end{bmatrix}^T$$

if $A = \begin{bmatrix} 1 & 5 & 2 \\ 0 & -1 & 2 \\ 0 & 0 & 1 \end{bmatrix}$

$$\therefore A^{-1} = \begin{bmatrix} -(-1) & (0) & 2(0) \\ (5) & -(1) & (0) \\ -(10+2) & (2) & -(-1) \end{bmatrix}^T$$

$$= \begin{bmatrix} 1 & 0 & 0 \\ 5 & -1 & 0 \\ -12 & 2 & 1 \end{bmatrix}^T$$

A conjugate or A^c

$\therefore A^{-1} = (A^c)^T$

$A A^{-1} = A^T A = I$

$\therefore$ if $AB = BA = I$ then $B = A^{-1}$

$$= \begin{bmatrix} 1 & 5 & -12 \\ 0 & -1 & 2 \\ 0 & 0 & 1 \end{bmatrix} = A^{-1}$$

④ Transpose :- Swapping rows and columns.

$A =$ | 1 | 2 | 3 | ⇒ 0
| 4 | 5 | 6 | ⇒ 1
2 × 3

then $A^T =$

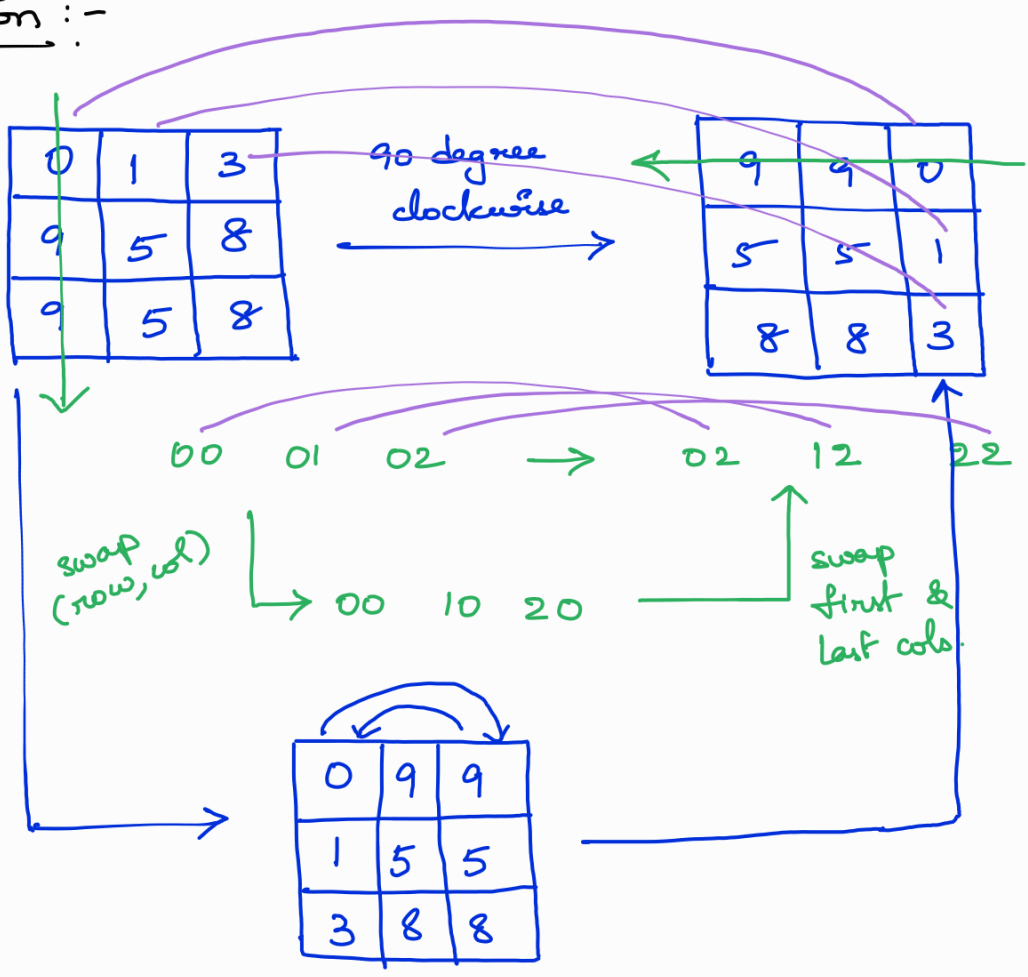
1	4
2	5
3	6

(0, 1 ↓) 3 × 2

Scalar multiplication :-

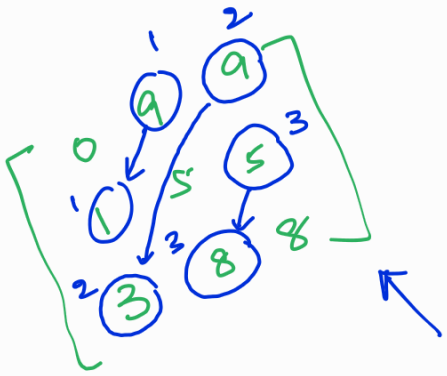
$$3 \times \begin{bmatrix} 1 & 2 & 3 \\ 1 & 0 & 4 \end{bmatrix} = \begin{bmatrix} 3 & 6 & 9 \\ 3 & 0 & 12 \end{bmatrix}$$

⑤ Rotation :-

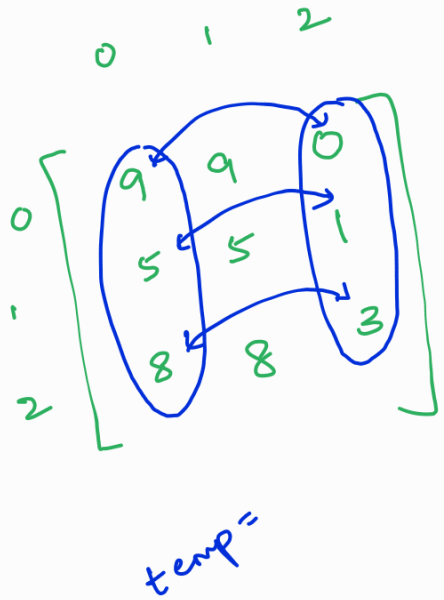


appl :- swap row, col & swap first & last col elements.

function (A[n][n]) {



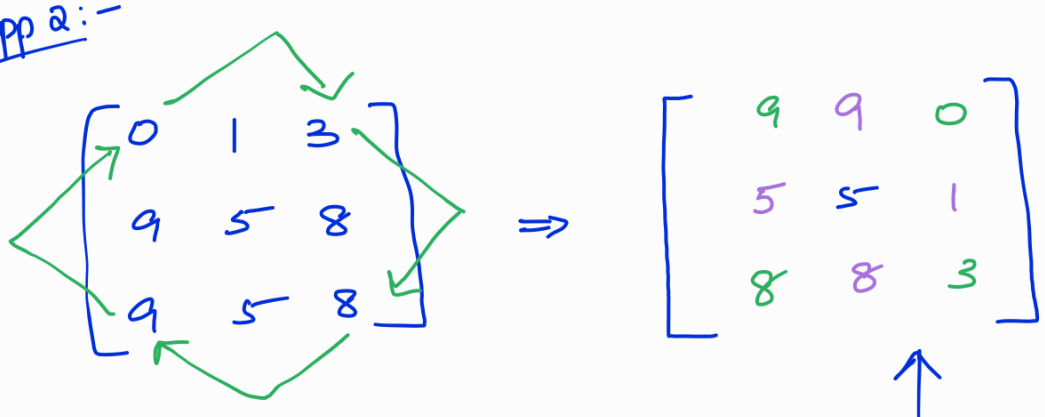

```
for (int i = 0; i < n ; i++) {
    for(int j = i+1; j < n ; j++) {
        temp = A[i][j]
        A[i][j] = A[j][i]
        A[j][i] = temp
    }
}
```

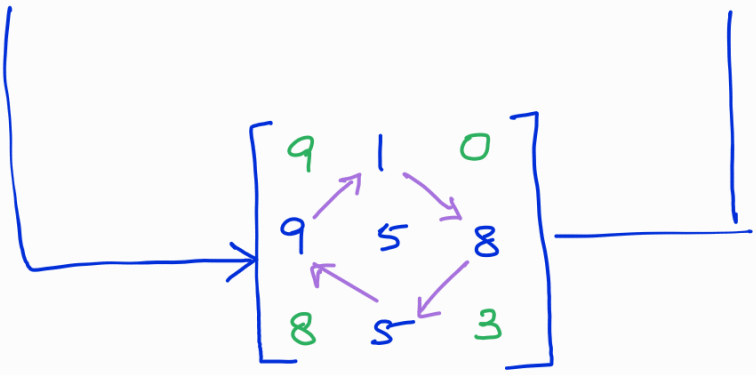



```
for (int j = 0; j < n/2 ; j++) {
    for(int i = 0; i < n; i++) {
        temp = A[i][j]
        A[i][j] = A[i][n-j-1]
        A[i][n-j-1] = temp.
    }
}
```

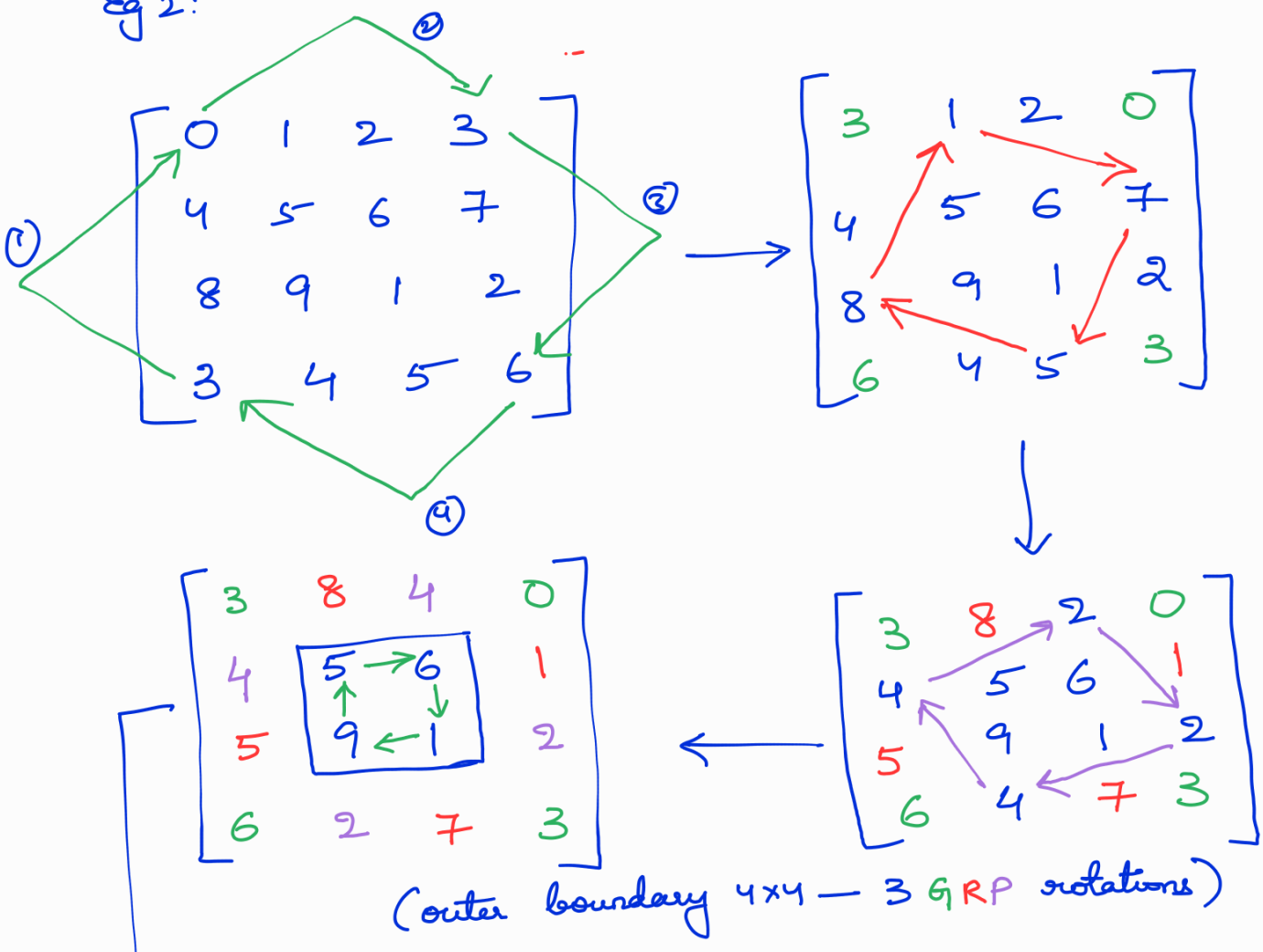
}

app 2:-



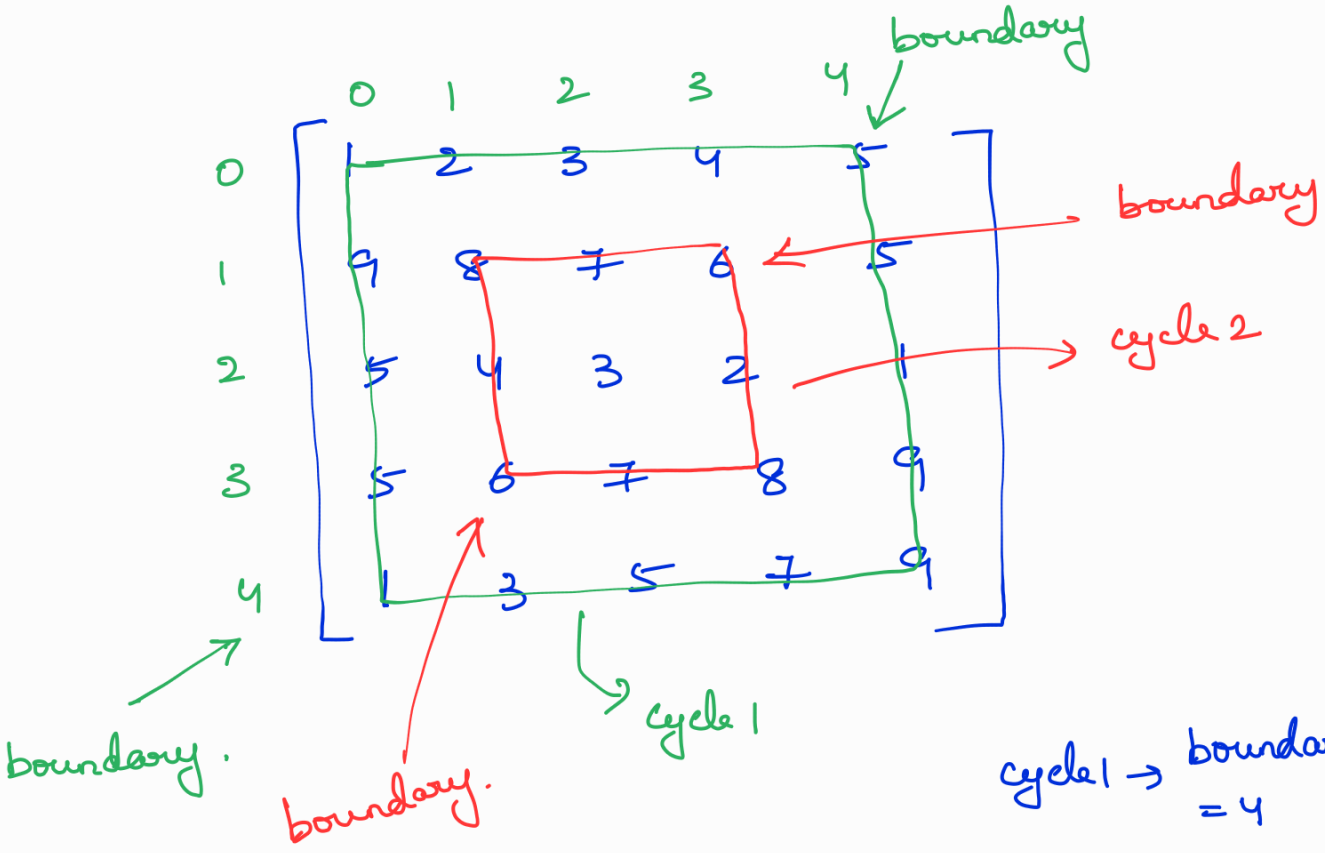


Eg 2:



(outer boundary 4×4 — 3 GRP rotations)

(inner boundary 2×2 — 1 GRP rotation)



cycle 1 → boundary = 4

cycle 2 → boundary = 3

each cycle is represented by its boundary.

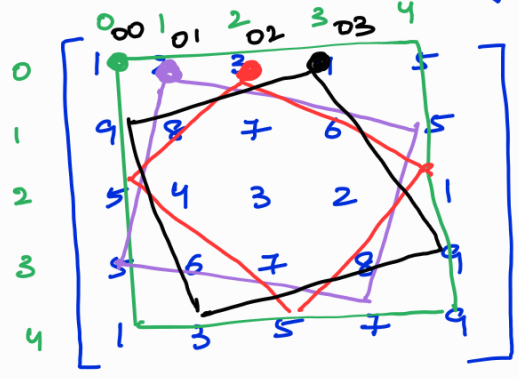
outer cycle is 5×5 and boundary = 4

inner cycle is 3×3 and boundary = 3

⇒ for every cycle completion, boundary decrements.

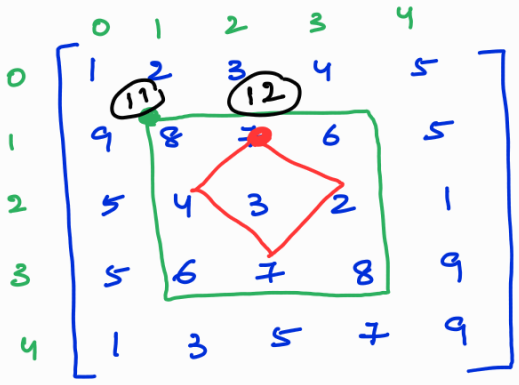
For outer cycle, there are 4 rotations starting at

{00, 01, 02, 03}



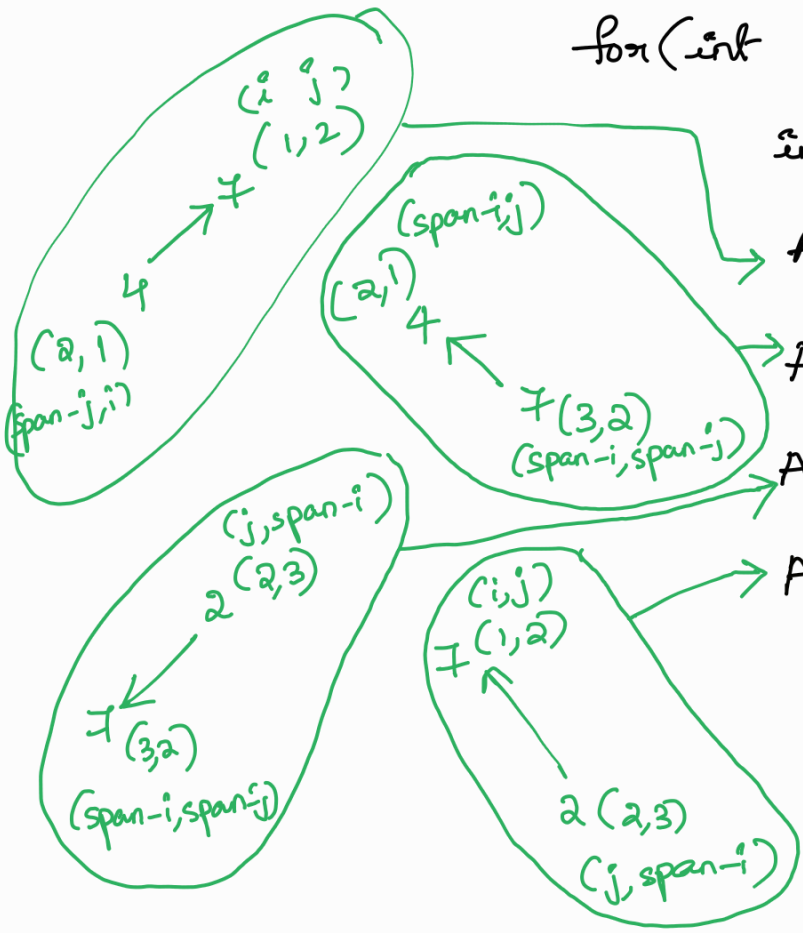
For inner cycle, there are 2 rotations starting at

{11, 12}



Code:-

```
int n = A.length;
int span = n-1, boundary = span;
for(int i=0; i < boundary; i++) {
    for(int j=i; i< boundary; j++) {
        int temp = A[i][j];
        A[i][j] = A[span-j][i];
        A[span-j][i] = A[span-i][span-j];
        A[span-i][span-j] = A[j][span-i];
        A[j][span-i] = temp;
```

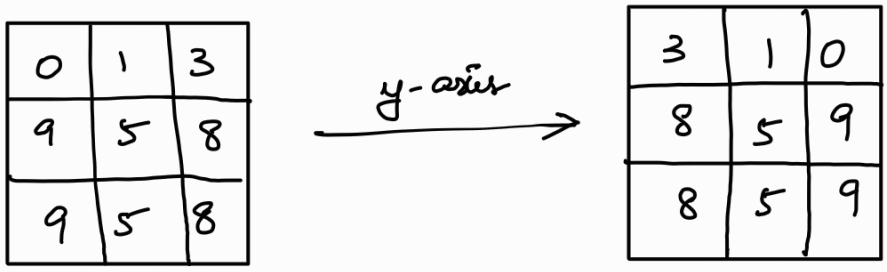


⑥ Reflection:

0	1	3
9	5	8
9	5	8

x-axis →

9	5	8
9	5	8
0	1	3



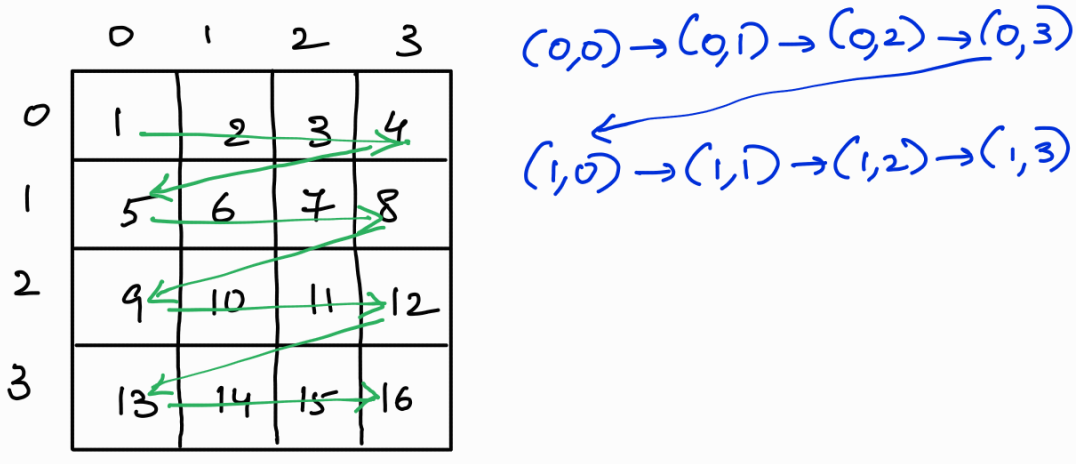
Matrix traversal :-

The process of systematically visiting each element in a matrix exactly once.

1. Row-major traversal.
2. Column-major traversal.
3. Diagonal traversal.
4. Spiral traversal

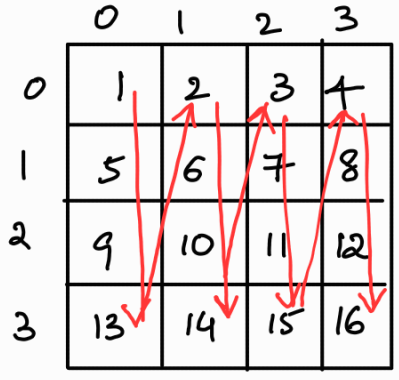
1. Row-major traversal :-

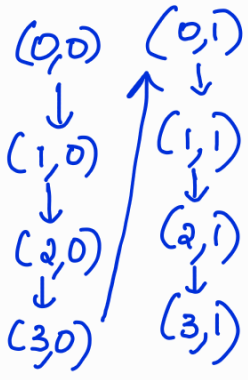
1. row by row
2. horizontally first and then vertically.



```
for(int row=0; row<n; row++) {
    for(int col=0; col<n; col++) {
        print arr[row][col];
    }
}
```

② Column-major traversal :-





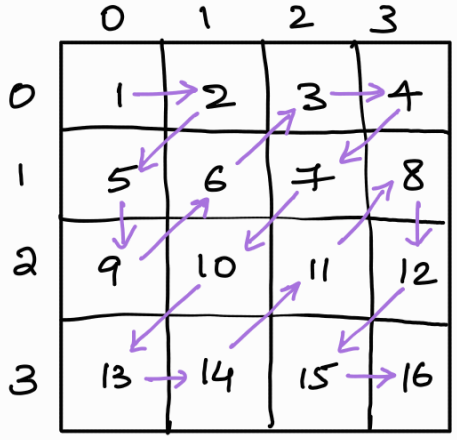
① opposite of row-major traversal

② traverse col-by-col.

```
for(int col=0; col<n; col++) {
    for(int row=0; row<n; row++) {
        print arr[row][col];
    }
}
```

③ Diagonal traversal :-

right
bottom-left } repeat
down
top-right

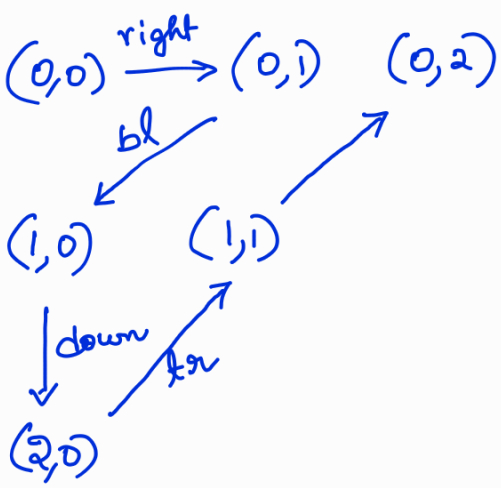


right → col++ (until col = n-1)

bottom-left → until col = 0 || row = n-1
col--
row++

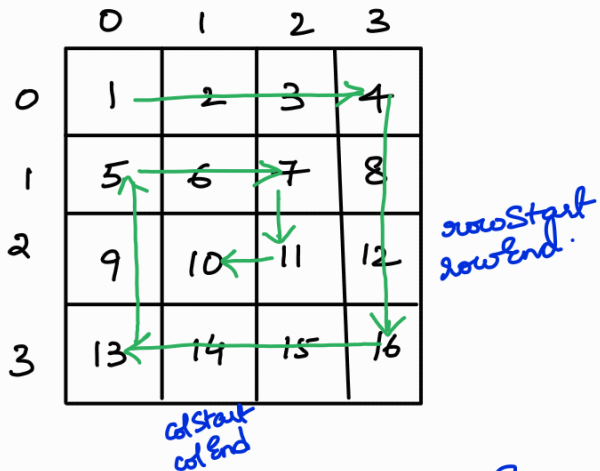
down → row++ (until row = n-1)

top-right → until col = n-1 || row = 0
col++
row--



④ Spiral traversal :-

right
down
left
up.
} repeat until all elements are visited.




```
while( rowStart <= rowEnd && colStart <= colEnd) {
  //right
  for ( j = colStart; j <= colEnd; j++ )
      array[rowStart][j];

  rowStart++;
  //Down
  for ( i = rowStart; i <= rowEnd; i++)
      array[i][colEnd];

  colEnd--;
  //left
  if (rowStart <= rowEnd)
    for( j = colEnd; j >= colStart; j--)
        array[rowEnd][j];

  rowEnd--;
```

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

```
// up
if (colStart <= colEnd)
    for(i = rowEnd; i >= rowStart; i--)
                array[i][colStart];

    colStart++;
}
```

Examples :-

Following are examples that illustrate problems that can be solved with the above approaches.

① Flip and invert an image :-

Input matrix

1	1	0
1	0	1
0	0	0

90° clockwise →

Flipped matrix

0	1	1
0	0	1
0	1	0

Flipped matrix

0	1	1
0	0	1
0	1	0

invert →

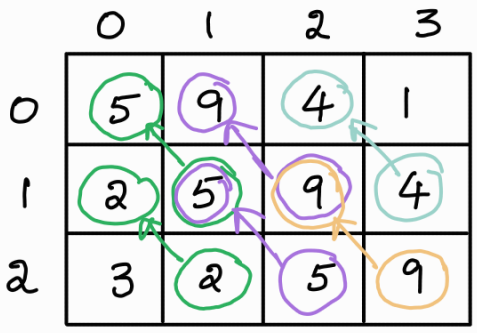
inverted matrix

1	0	0
1	1	0
1	0	1

② Toeplitz matrix :-

Given a m x n matrix, return TRUE if the matrix is Toeplitz, otherwise, return FALSE.

A matrix is Toeplitz if every descending diagonal from left to right has the same elements.



① Start traversal from second row and second column.
$i = 1, j = 1$.

② For each element, matrix[i][j], we'll check if it is equal to top-left, matrix[i-1][j-1]. If true, continue traversal, otherwise return FALSE.

[1][1] — [0][0] ✓

[1][2] — [0][1] ✓

[1][3] — [0][2] ✓

[2][1] — [1][0] ✓

[2][2] — [1][1] ✓

[2][3] — [1][2] ✓

$i = 1, j = 1$

```
while( i <= rowSize && j <= colSize) {
    if(arr[i][j] == arr[i-1][j-1]) {
```

```
            if (j == colSize-1) {
                i++;
                j=1;
            } else {
                return false;
            }
        }
    }
    return true;
```

Real-World Problems:-

① **Image Processing**:- Images where each pixel's color values are stored in a matrix.

Matrix transformations such as scaling, rotation, translation, and affine transformations are applied to manipulate images in graphics software.

② **Computer graphics and gaming**:- Matrices are used to represent transformations such as translating, rotating, and scaling objects in 3D space. They are also used for transformations of vertices in 3D graphics rendering pipelines, essential for creating realistic scenes in video games and simulations.

③ **Data analysis and statistics**:- Matrices are used in statistics for representing data sets. They are used in techniques like linear regression, principal component analysis (PCA) and factor analysis. Also in multivariate analysis, covariance matrices and correlation matrices.

④ **Machine Learning** :- Matrices are central to many machine learning algorithms, especially in tasks like linear regression, logistic regression, support vector machines (SVM), neural networks, and dimensionality reduction techniques.

PRACTICE PROBLEMS

1)

73. Set Matrix Zeroes

Medium | Topics | Companies | Hint

Given an `m x n` integer matrix `matrix`, if an element is `0`, set its entire row and column to `0`'s.

You must do it in place.

Example 1:

1	1	1
1	0	1
1	1	1

⇒

1	0	1
0	0	0
1	0	1

```
Input: matrix = [[1,1,1],[1,0,1],[1,1,1]]
Output: [[1,0,1],[0,0,0],[1,0,1]]
```

Example 2:

0	1	2	0
3	4	5	2
1	3	1	5

⇒

0	0	0	0
0	4	5	0
0	3	1	0

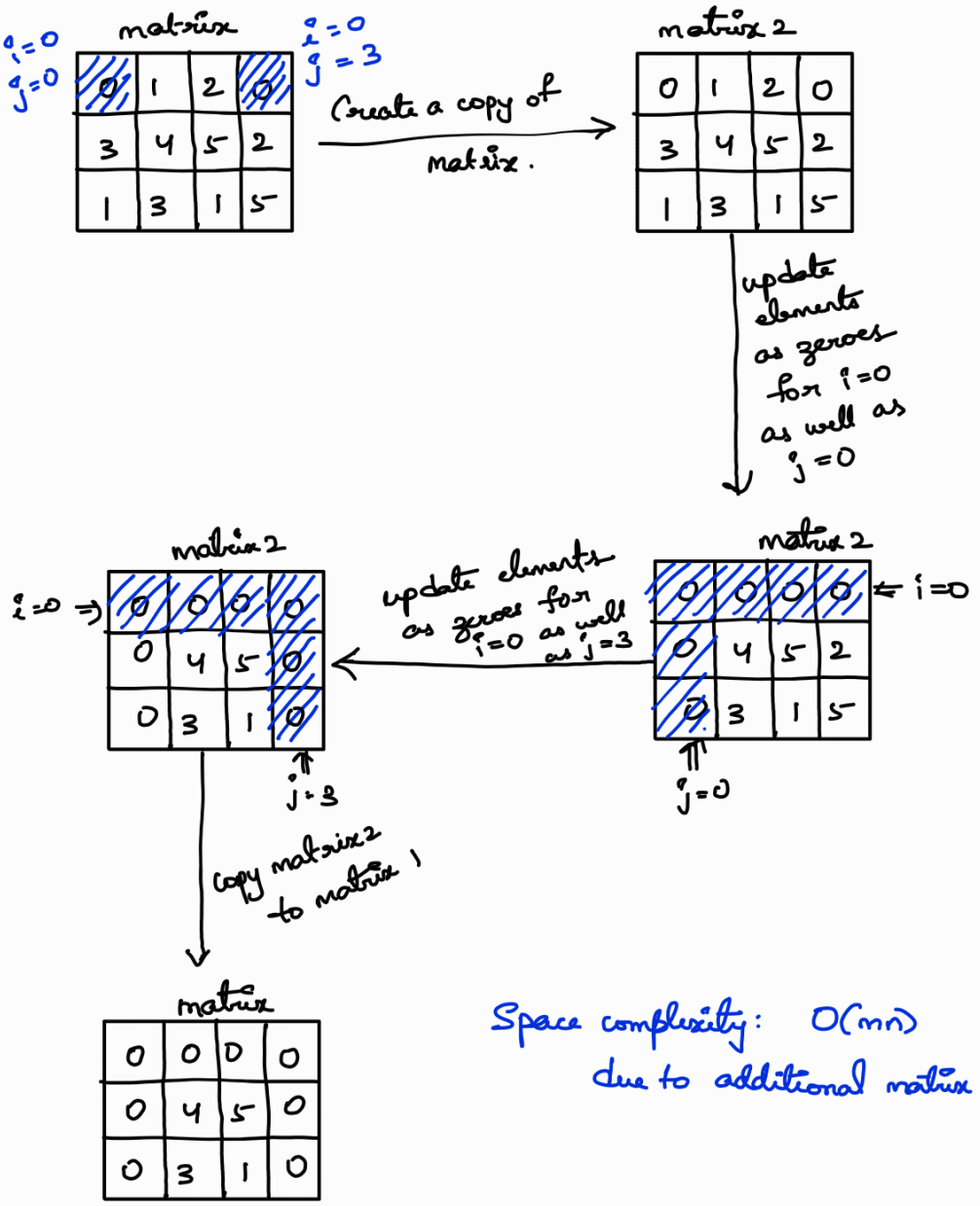
```
Input: matrix = [[0,1,2,0],[3,4,5,2],[1,3,1,5]]
Output: [[0,0,0,0],[0,4,5,0],[0,3,1,0]]
```

Constraints:

- `m == matrix.length`
- `n == matrix[0].length`
- `1 <= m, n <= 200`
- -2^{31} `<= matrix[i][j] <=` $2^{31} - 1$

Approach 1 :-

Brute force :-



Space complexity: O(mn) due to additional matrix

Code for approach 1 :-

```java
public void setZeroes(int[][] matrix) {
    int m = matrix.length, n = matrix[0].length;
    int[][] matrix2 = new int[m][n];
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            matrix2[i][j] = matrix[i][j];
        }
    }
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (matrix[i][j] == 0) {
                for (int k = 0; k < n; k++) {
                    matrix2[i][k] = 0;
                }
                for (int k = 0; k < m; k++) {
                    matrix2[k][j] = 0;
                }
            }
        }
    }

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            matrix[i][j] = matrix2[i][j];
        }
    }
}
```

$m*n$

$m+n$

$(m*n)(m+n)$

∴ Time complexity:

$O(mn(m+n))$

Approach 2 :-

```java
public void setZeroes(int[][] matrix) {
    int m = matrix.length, n = matrix[0].length;
    int[] rows = new int[m];
    int[] cols = new int[n];
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (matrix[i][j] == 0) {
                rows[i] = 1;
                cols[j] = 1;
            }
        }
    }

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (rows[i] == 1 || cols[j] == 1)
                matrix[i][j] = 0;
        }
    }
}
```

① Create rows array and cols array to store the rows and cols where 0 is present while traversing the matrix

② Traverse matrix again and if the row i & column j has 0 (check for rows[i] & cols[j] = 1) then update the element at i,j to 0.

① $O(m*n)$ ② $O(m*n)$ ⇒ Time complexity: $O(mn)$.

② rows (size m) cols (size n) ⇒ Space complexity: $O(m+n)$.

Follow up:

- A straightforward solution using `O(mn)` space is probably a bad idea.
- A simple improvement uses `O(m + n)` space, but still not the best solution.
- Could you devise a constant space solution?

Approach 3:- Constant space solution.

	0	1	2	3
0	0	1	2	0
1	2	4	0	7
2	6	28	1	2
3	1	2	3	2

As we start looking at row 0 and col 0, if any zeroes are present, we cannot update the corresponding rows and columns to zeroes right away. Because, we will be unable to identify original zero positions.

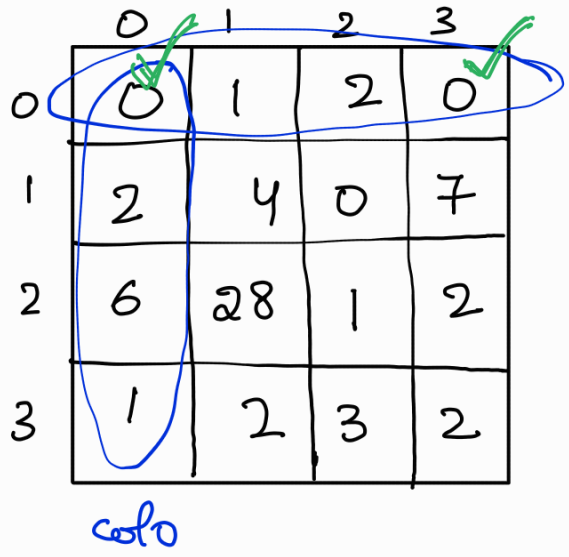
Ex:-

	0	1	2	3
0	0	0	0	0
1	0	4	0	0
2	0	28	1	0
3	0	2	3	0

→ original zero position

→ updated zeroes.

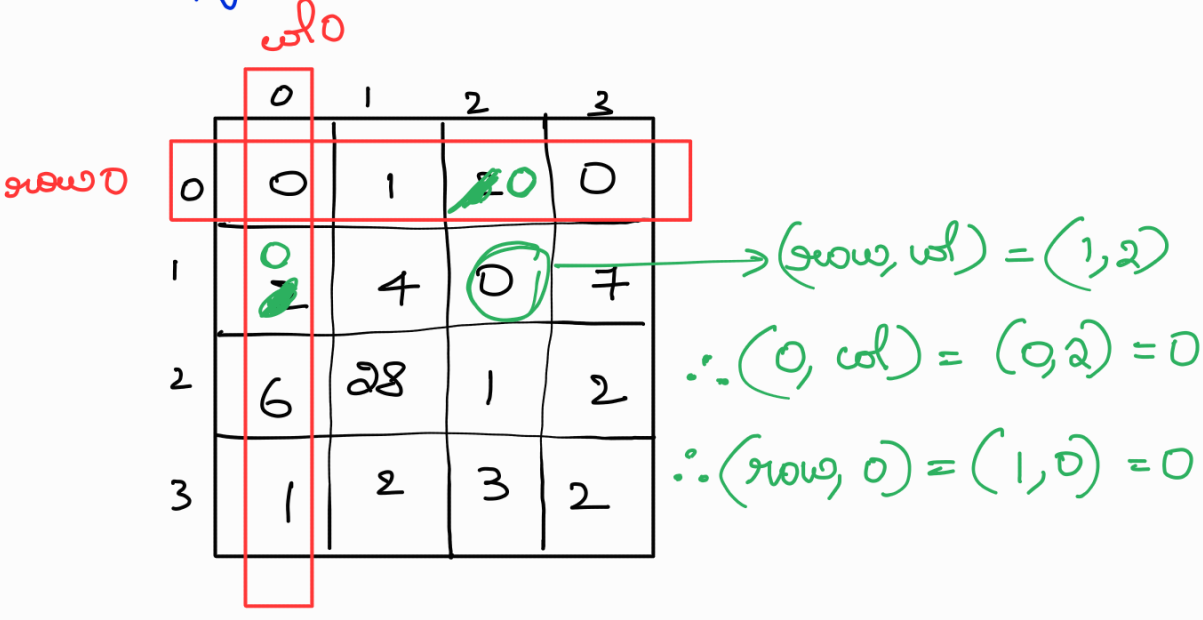
Step (1): Let's save row 0 and col 0, zeroes information in boolean flags hasZeroRow, hasZeroCol and use them in the end.



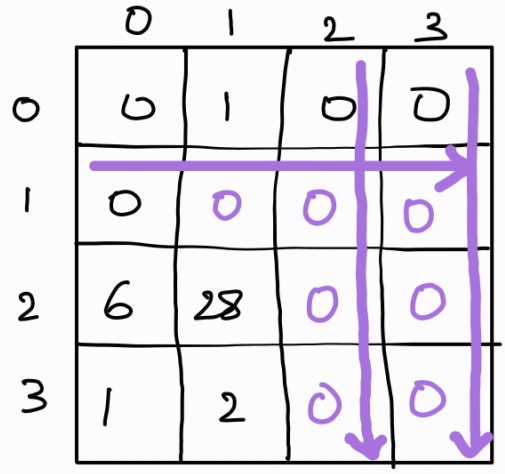
For example,
row0 has 0
col0 has 0
∴ hasZeroRow = true
∴ hasZeroCol = true.

Step ②: Traverse matrix except row0, col0 (call it mini-matrix) as row0 & col0 are already done, starting from (1,1)

If zero is present at any (row, col) position, update the same "col" in row 0 and same "row" in col0. See fig;



Step ③: Let's update mini-matrix, starting from (1,1), with zeroes if corresponding row (matrix[i][0]) or col (matrix[0][j]) is 0.

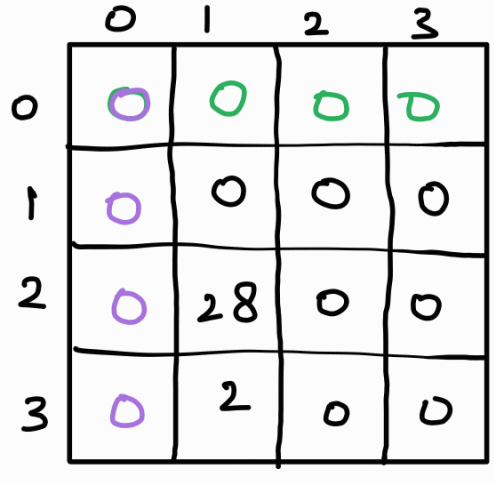


In the fig., (1,0) = 0, so (1,1), (1,2), (1,3) = 0

(0,2) = 0, so (1,2), (2,2), (3,2) = 0

(0,3) = 0, so (1,3)(2,3)(3,3) = 0.

Step (4): Update row0 and col0, if it has zeroes (hasRowZero = true, hasColZero = true)



hasRowZero = true.

hasColZero = true

Thoughts :-

Why do we need to update zeroes of mini-matrix and row0, col0 separately?

Ans:- Another Example:

Let's consider same matrix but without zeroes in row and col.

	0	1	2	3
0	8	1	2	4
1	2	4	0	7
2	6	28	1	2
3	1	2	3	2

Expected answer:

	0	1	2	3
0	8	1	0	4
1	0	0	0	0
2	6	28	0	2
3	1	2	0	2

Step ①:

	0	1	2	3
0	8	1	2	4
1	2	4	0	7
2	6	28	1	2
3	1	2	3	2

hasZeroRow = false

hasZeroCol = false.

Step ②: Mini-matrix (starting at (1,1))

	0	1	2	3
0	8	1	0	4
1	0	4	0	7
2	6	28	1	2
3	1	2	3	2

(1,2) = 0.

∴ set (0,2) = 0

set (1,0) = 0

Step ③:

What if you update full matrix instead of mini-matrix with zeroes?

	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	0	28	0	2
3	0	2	0	2

This is not expected answer.

whereas, in previous example, you can update full matrix in step ③ to get expected answer

	0	1	2	3
0	0	1	2	0
1	2	4	0	7
2	6	28	1	2
3	1	2	3	2

Given

	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	0	28	0	0
3	0	2	0	0

Expected

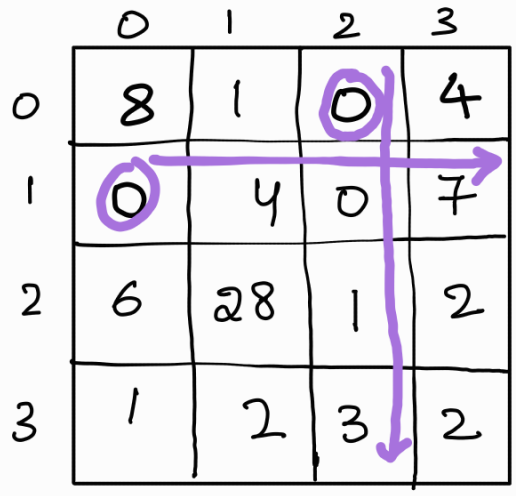
	0	1	2	3
0	0	1	0	0
1	0	4	0	7
2	6	28	1	2
3	1	2	3	2

Step ②

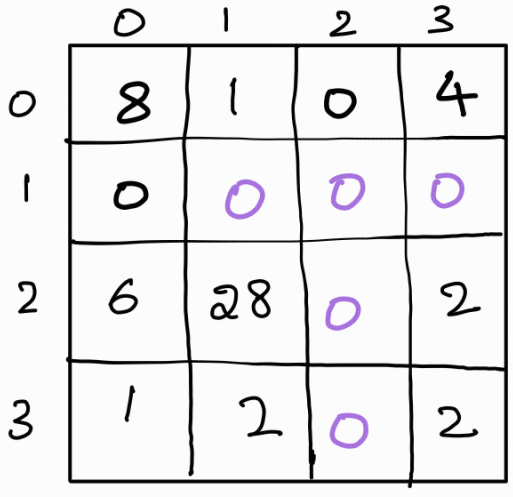
	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	0	28	0	0
3	0	2	0	0

Step ③ update full matrix with zeroes.

Hence, we need to update only mini-matrix (starting from (1,1)) in this step.



	0	1	2	3
0	8	1	0	4
1	0	4	0	7
2	6	28	1	2
3	1	2	3	2



	0	1	2	3
0	8	1	0	4
1	0	0	0	0
2	6	28	0	2
3	1	2	0	2

Step (4): hasZeroRow = false

hasZeroCol = false.

So we are not updating row 0 and col 0.

Hence ans is:

8	1	0	4
0	0	0	0
6	28	0	2
1	2	0	2

(which is expected ☺)

```java
public void setZeroes(int[][] matrix) {

    boolean hasRowZero = false;
    boolean hasColZero = false;

    int m = matrix.length;
    int n = matrix[0].length;

    for (int i = 0; i < m; i++) {
        if (matrix[i][0] == 0) {
            hasColZero = true;
            break;
        }
    }

    for (int j = 0; j < n; j++) {
        if (matrix[0][j] == 0) {
            hasRowZero = true;
            break;
        }
    }

    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            if (matrix[i][j] == 0) {
                matrix[i][0] = 0;
                matrix[0][j] = 0;
            }
        }
    }

    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            if (matrix[0][j] == 0 || matrix[i][0] == 0) {
                matrix[i][j] = 0;
            }
        }
    }

    if (hasRowZero) {
        for (int j = 0; j < n; j++) {
            matrix[0][j] = 0;
        }
    }

    if (hasColZero) {
        for (int i = 0; i < m; i++) {
            matrix[i][0] = 0;
        }
    }
}
```

step ①

step ②

step ③

step ④

We can further reduce the lines of code as follows:

```java
public void setZeroes(int[][] matrix) {

    boolean hasRowZero = false;
    boolean hasColZero = false;

    int m = matrix.length;
    int n = matrix[0].length;

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (matrix[i][j] == 0) {
                if (j == 0) hasColZero = true;
                if (i == 0) hasRowZero = true;
                matrix[0][j] = 0;
                matrix[i][0] = 0;
            }
        }
    }

    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            if (matrix[0][j] == 0 || matrix[i][0] == 0) {
                matrix[i][j] = 0;
            }
        }
    }

    if (hasRowZero) {
        for (int j = 0; j < n; j++) {
            matrix[0][j] = 0;
        }
    }

    if (hasColZero) {
        for (int i = 0; i < m; i++) {
            matrix[i][0] = 0;
        }
    }
}
```

combined step ① & step ②.

2)

1706. Where Will the Ball Fall

Medium | Topics | Companies | Hint

You have a 2-D `grid` of size `m x n` representing a box, and you have `n` balls. The box is open on the top and bottom sides.

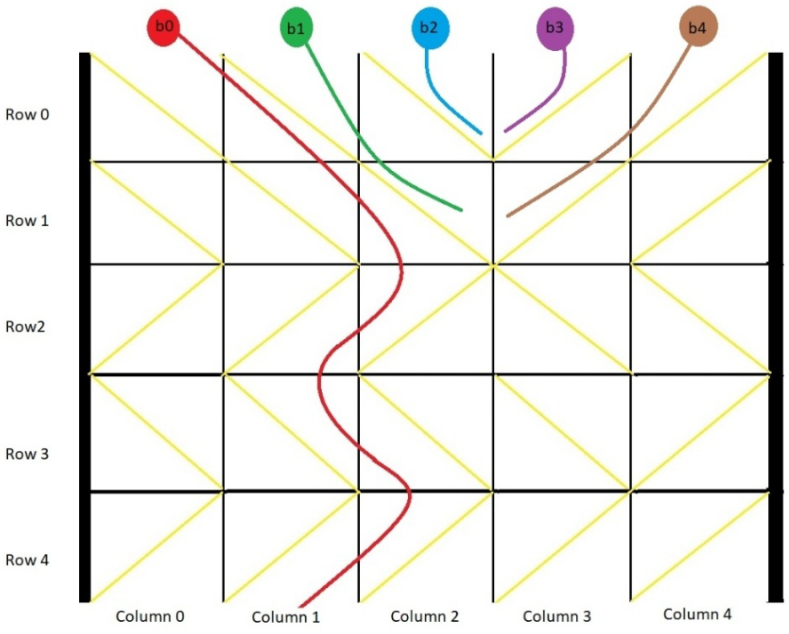
Each cell in the box has a diagonal board spanning two corners of the cell that can redirect a ball to the right or to the left.

- A board that redirects the ball to the right spans the top-left corner to the bottom-right corner and is represented in the grid as `1`.
- A board that redirects the ball to the left spans the top-right corner to the bottom-left corner and is represented in the grid as `-1`.

We drop one ball at the top of each column of the box. Each ball can get stuck in the box or fall out of the bottom. A ball gets stuck if it hits a "V" shaped pattern between two boards or if a board redirects the ball into either wall of the box.

Return *an array* `answer` *of size* `n` *where* `answer[i]` *is the column that the ball falls out of at the bottom after dropping the ball from the* i^{th} *column at the top, or* `-1` *if the ball gets stuck in the box.*

Example 1:



> **Input:** grid = [[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]]
> **Output:** [1,-1,-1,-1,-1]
> **Explanation:** This example is shown in the photo.
> Ball b0 is dropped at column 0 and falls out of the box at column 1.
> Ball b1 is dropped at column 1 and will get stuck in the box between column 2 and 3 and row 1.
> Ball b2 is dropped at column 2 and will get stuck on the box between column 2 and 3 and row 0.
> Ball b3 is dropped at column 3 and will get stuck on the box between column 2 and 3 and row 0.
> Ball b4 is dropped at column 4 and will get stuck on the box between column 2 and 3 and row 1.

Example 2:

> **Input:** grid = [[-1]]
> **Output:** [-1]
> **Explanation:** The ball gets stuck against the left wall.

Example 3:

> **Input:** grid = [[1,1,1,1,1,1],[-1,-1,-1,-1,-1,-1],[1,1,1,1,1,1],[-1,-1,-1,-1,-1,-1]]
> **Output:** [0,1,2,3,4,-1]

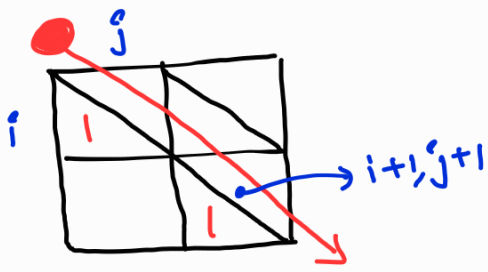
Constraints:

- `m == grid.length`
- `n == grid[i].length`
- `1 <= m, n <= 100`
- `grid[i][j]` is `1` or `-1`.

Approach:-

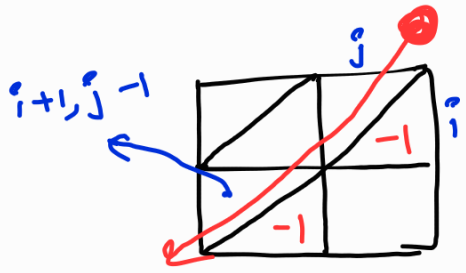
point ① The approach is pretty straightforward.

point ② Ball can reach down the slope if



$(i,j) = 1$ && $(i, j+1) = 1$

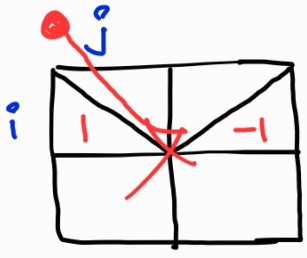
next position of ball $(i+1, j+1)$



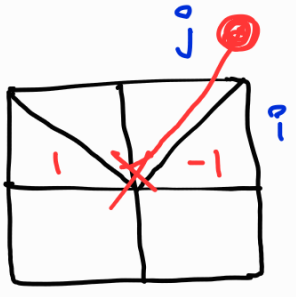
$(i,j) = -1$ && $(i, j-1) = -1$

next position of ball $(i+1, j-1)$

point ③ Ball cannot reach down the slope if

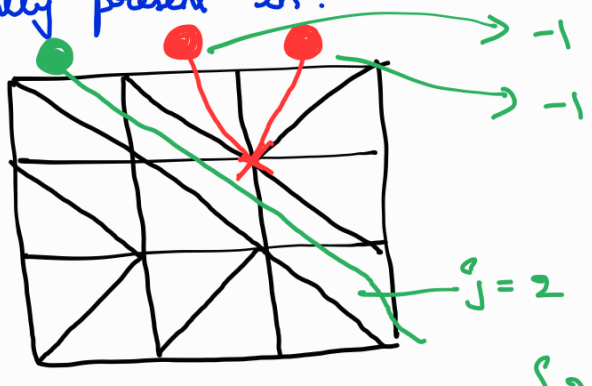


$(i,j) = 1$ && $(i, j+1) = -1$



$(i,j) = 1$ && $(i, j-1) = 1$

Point ④ When ball reaches last row, return the col no. of the cell the ball currently present in.



ans: {2, -1, -1}

Code in recursion:

```java
class Solution {
    public int[] findBall(int[][] grid) {

        int m = grid[0].length;
        int[] arr = new int[m];
        for (int i = 0; i < m; i++) {
            arr[i] = dfs(grid, 0, i);
        }
        return arr;
    }

    public int dfs (int[][] grid, int i, int j) {
        if (i == grid.length) {
            return j;
        }

        if (j < 0 || j >= grid[0].length) {
            return -1;
        }

        if (grid[i][j] == 1 && j + 1 < grid[0].length && grid[i][j+1] == 1) {
            return dfs (grid, i + 1, j +1);
        } else if (grid[i][j] == -1 && j - 1 >= 0 && grid[i][j - 1] == -1) {
            return dfs (grid, i + 1, j -1);
        }

        return -1;
    }
}
```

} n times.

→ max m times.

TC:-

$O(mn)$

m = no. of rows.

n = no. of columns.

SC:-

$O(1)$

(not considering output space)

Code in iteration:

①

```java
    public int[] findBall(int[][] grid) {

        int m = grid[0].length;
        int row = 0;
        int[] ans = new int[m];
        for (int col = 0; col < m; col++) {
            int i = row, j = col;
            while(i < grid.length) {

                if (grid[i][j] == 1 && j+1 < m && grid[i][j+1] == 1) {
                    i++;
                    j++;
                } else if(grid[i][j] == -1 && j-1 >=0 && grid[i][j-1] == -1) {
                    i++;
                    j--;
                } else {
                    break;
                }
            }
            ans[col] = i == grid.length ? j : -1 ;

        }
        return ans;
    }
```

(2)

```java
public int[] findBall(int[][] grid) {

    int m = grid.length;
    int n = grid[0].length;
    int[] ans = new int[n];
    for (int col = 0; col < n; col++) {
        int j1 = col, j2;
        for (int row = 0; row < m; row++) {
            j2 = j1 + grid[row][j1];
            if (j2 < 0 || j2 >= n || grid[row][j2] != grid[row][j1]) {
                j1 = -1;
                break;
            }
            j1 = j2;
        }
        ans[col] = j1;

    }
    return ans;
}
```

(3)

289. Game of Life

Medium | Topics | Companies

According to Wikipedia's article: "The **Game of Life**, also known simply as **Life**, is a cellular automaton devised by the British mathematician John Horton Conway in 1970."

The board is made up of an `m x n` grid of cells, where each cell has an initial state: **live** (represented by a `1`) or **dead** (represented by a `0`). Each cell interacts with its eight neighbors (horizontal, vertical, diagonal) using the following four rules (taken from the above Wikipedia article):

1. Any live cell with fewer than two live neighbors dies as if caused by under-population.

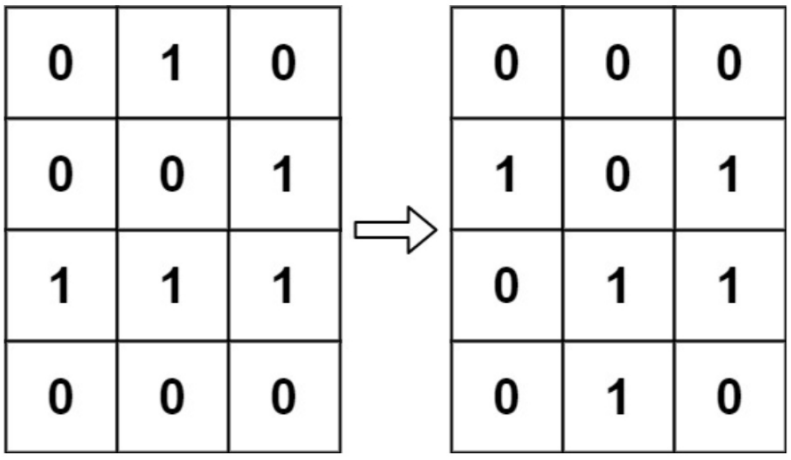
2. Any live cell with two or three live neighbors lives on to the next generation.

3. Any live cell with more than three live neighbors dies, as if by over-population.

4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

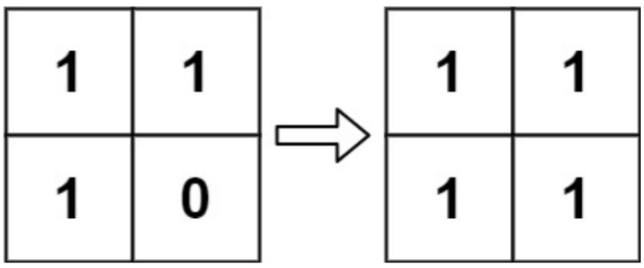
The next state is created by applying the above rules simultaneously to every cell in the current state, where births and deaths occur simultaneously. Given the current state of the `m x n` grid `board`, return *the next state*.

Example 1:




```
Input: board = [[0,1,0],[0,0,1],[1,1,1],[0,0,0]]
Output: [[0,0,0],[1,0,1],[0,1,1],[0,1,0]]
```

Example 2:



```
Input: board = [[1,1],[1,0]]
Output: [[1,1],[1,1]]
```

Constraints:

- `m == board.length`
- `n == board[i].length`
- `1 <= m, n <= 25`
- `board[i][j]` is `0` or `1`.

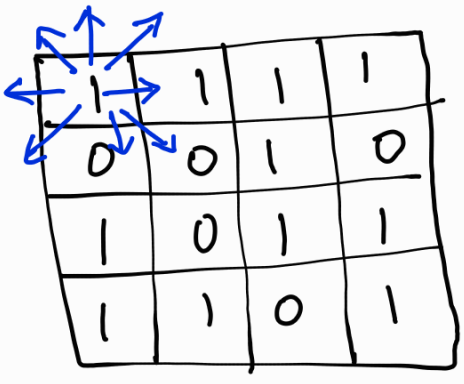
Follow up:

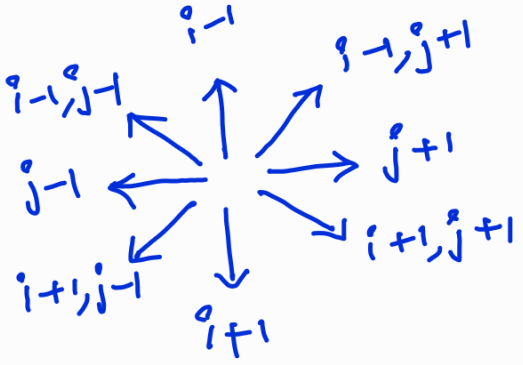
- Could you solve it in-place? Remember that the board needs to be updated simultaneously: You cannot update some cells first and then use their updated values to update other cells.
- In this question, we represent the board using a 2D array. In principle, the board is infinite, which would cause problems when the active area encroaches upon the border of the array (i.e., live cells reach the border). How would you address these problems?

Approach 1:-

We create a temporary (temp) matrix of same input size.

We will traverse each element of board and calculate total neighbours.





After calculating sum, based on given input criteria, we will update the element but in temp matrix.

Later we copy temp matrix to board.

```java
class Solution {
    public void gameOfLife(int[][] board) {

        int m = board.length, n = board[0].length;

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

        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                int neighbors = calculateNeighbors(board, i, j);
                if (board[i][j] == 1) {
                    temp[i][j] = (neighbors < 2 || neighbors > 3) ? 0 : 1;

                } else {
                    temp[i][j] = neighbors == 3 ? 1: 0;
                }
            }
        }

        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                board[i][j] = temp[i][j];
            }
        }
    }

    public int calculateNeighbors(int[][] board, int row, int col) {
        int sum = 0;
        for (int i = Math.max(row - 1, 0); i <= Math.min(row + 1, board.length - 1); i++) {
            for (int j = Math.max(col - 1, 0); j <= Math.min(col + 1, board[0].length - 1); j++) {
                sum += board[ i][ j];

            }
        }
        sum -= board[row][col];
        return sum;
    }
}
```

TC: O(mn)

SC: O(mn)

Approach:- in-place:-

represent current and next states using bits in units place and tens place.

dead ⇒ 0

alive ⇒ 1

We have four states:

dead ← dead ⇒ 00 → 0

dead ← alive ⇒ 01 → 1

alive ← dead ⇒ 10 → 2

alive ← alive ⇒ 11 → 3

(next) (current)

In the given input, the current states are either 0 or 1 ⇒ 00 and 01 by default which means next state is by default "dead". So, the scenarios that make next state "dead" can be ignored (i.e., no operation necessary to change next state to 0 as it is already zero)

We are left with operations that make 0 and 1 (& 00 and 01) alive for next state (i.e., 10 and 11)

For each cell's LSB, check the 8 pixels around itself, and set the cell's MSB.

⇒ 01 → 11 : when board[i][j] = 1 and neighbours count is (>= 2 && <= 3)

⇒ 00 → 10 : when board[i][j] == 0 and neighbours count is (== 3)

* To get current state, get LSB.
  LSB can be retrieved by doing (board[i][j] & 1)
  To get the next state, get MSB (or second bit)
  MSB can be retrieved by doing (board[i][j] >> 1)

To understand this:

Ex 1 - 2 ⇒ $(10)_2$

To get first bit, 2 & 1 ⇒ 10 & 1

$$\begin{array}{r} 10 \\ \&\ 01 \\ \hline (00)_2 \end{array}$$

$$= (0)_{10}$$

current state = 0

To get second bit $2 >> 1 \Rightarrow (10)_2 >> 1$

(i.e., shift digits to the right by 1 place)

$10 >> 1$

$= (01)_2$

$= (1)_{10}$

next state = 1.

```java
public void gameOfLife(int[][] board) {

    int m = board.length, n = board[0].length;

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            int neighbors = calculateNeighbors(board, i, j);
            if (board[i][j] == 1 && neighbors >= 2 && neighbors <= 3) {
                board[i][j] = 3;// 01 --> 11
            } else if (board[i][j] == 0 && neighbors == 3){
                board[i][j] = 2; // 00 ---> 10
            }
        }
    }

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            board[i][j] >>= 1; // update board with new state
        }
    }
}

public int calculateNeighbors(int[][] board, int row, int col) {
    int neighbors = 0;
    for (int i = Math.max(row - 1, 0); i <= Math.min(row + 1, board.length - 1); i++) {
        for (int j = Math.max(col - 1, 0); j <= Math.min(col + 1, board[0].length - 1); j++) {
            neighbors += board[ i][ j] & 1;
        }
    }
    neighbors -= board[row][col]&1;
    return neighbors;
}
```

TC: O(mn)

SC: O(1)

3) Find pairs with given sum such that elements of pair are in different rows.

Example:

Input:

mat[4][4] = {{1, 3, 2, 4},
{5, 8, 7, 6},
{9, 10, 13, 11},
{12, 0, 14, 15}}

sum = 11

Output: (1, 10), (3, 8), (2, 9), (4, 7), (11, 0)

APPROACH 1:

```
pairSum (int[][] mat, int sum) {
    int m = mat.length;
    int n = mat[0].length;
    int count = 0;
    for(int i=0; i<m; i++){
        for(int k=i+1; k<m; k++) {
            for(int j=0; j<n; j++) {
                for(int l=0; l<n; l++) {
                    if(mat[i][j] + mat[k][l] == sum){
                        count++;
                    }
                }
            }
        }
    }
    return count;
}
```

TC: $O(m^2n^2)$

SC: $O(1)$.

APPROACH 2 :- (Sorting)

$$\begin{bmatrix} 1 & 3 & 2 & 4 \\ 5 & 8 & 7 & 6 \\ 9 & 10 & 13 & 11 \\ 12 & 0 & 14 & 15 \end{bmatrix}$$

sum = 11

sort all rows:

left

$$\begin{bmatrix} \textcircled{1} & 2 & 3 & 4 \\ 5 & 6 & 7 & \textcircled{8} \\ 9 & 10 & 11 & 13 \\ 0 & 12 & 14 & 15 \end{bmatrix}$$

i → , j → , right

Finding sum between two sorted rows:-

left left left left left

1 2 3 4

right right

5 6 7 8

1+8 = 9 < 11 left++

2+8 = 10 < 11 left++

3+8 = 11 = 11 left++, right-- (3, 8)

4+7 = 11 = 11 left++, right-- (4, 7)

```
pairSum(int[][] mat, int sum) {
    int n = mat.length; int count = 0;
    for(int i=0; i<n; i++) {
        Arrays.sort(mat[i]);   → n^2 log n
```

n × (n log n)

(n rows) × (sort each row)

```
for(int i=0; i<n-1; i++) {
    for(int j=i+1; j<n; j++) {
        int left=0, right=n-1;
        while(left<n && right>=0){
            int s = mat[i][left] +
                    mat[j][right]

            if(s == sum) {
                count++;
                left++;
                right--;
            } else if (s < sum) {
                left++;
            } else {
                right--;
            }
        }
    }
}
return count;
}
```

TC:- $O(n^2 \log n + n^3)$

SC:- $O(1)$

APPROACH 3:- (HASHING)

We store elements and their corresponding row number. Traverse the matrix again to check whether s-th pair exists. If exists, row numbers should be different. If

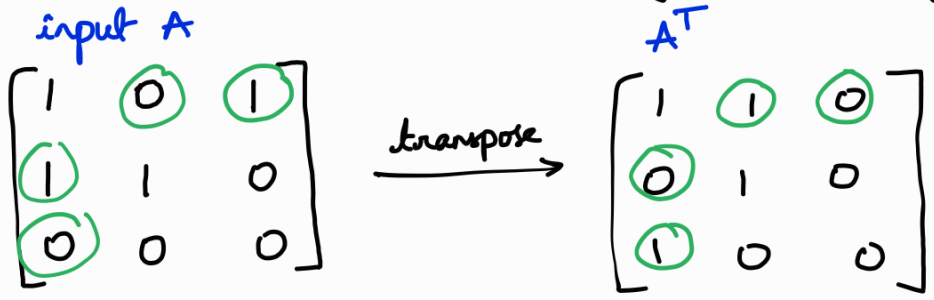
different count the pair.

```
pairSum(int[][] mat, int sum) {
    int n = mat.length, count = 0;
    Map<Integer, Integer> hm = new HashMap<>();
    for(int i=0; i<n; i++) {
        for(int j=0; j<n; j++) {
            int remSum = sum - mat[i][j];
            if(hm.containsKey(remSum)) {
                int row = hm.get(remSum);
                if(row > i) {
                    count++;
                }
            }
        }
    }
    return count;
}
```

TC: $O(n^2)$

SC: $O(n^2)$

4) Minimum flips to make binary matrix symmetric.



A matrix A is said to be symmetric if $A = A^T$

A transpose of a matrix will have same diagonal as the matrix.

In the above representation (1,0) becomes (0,1) in transpose. Similarly,

$(1,0) \rightarrow (0,1)$
$(2,0) \rightarrow (0,2)$
$(2,1) \rightarrow (1,2)$
} for a 3×3 matrix.

If we compare the above elements, we need 2 flips to make A and A^T equal. How?

Step①:

$$A: \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad A^T: \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

Flip (0,1) & (0,2) as (0,1), (0,2) of A & A^T different.

$$A: \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \longrightarrow A^T: \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Thus A and A^T are equal.

The key takeway from this approach is,

* we compared the row of the matrix with its column.
* we do not need to flip diagonal elements as in symmetry diagonal is same but only upper and lower triangular matrix varies.

Another example:-

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

→ mismatch.

(0,1) – (1,0) ✓
(0,2) – (2,0) ✓
(1,2) – (2,1) ✗

↓ flip (2,1)

$$A = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} \xrightarrow{\text{transpose}} A^T = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

$\therefore$ A is symmetric.

Code :-

```
minimumFlips (int[][] arr) {
    int n = arr.length;
    int count = 0;
    for(int i=0; i<n; i++) {
        for(int j=1; j<n; j++) {
            if(i!=j && arr[i][j] != arr[j][i]) {
                count++;
            }
        }
    }
    return count;
}
```

5)

2128. Remove All Ones With Row and Column Flips Premium

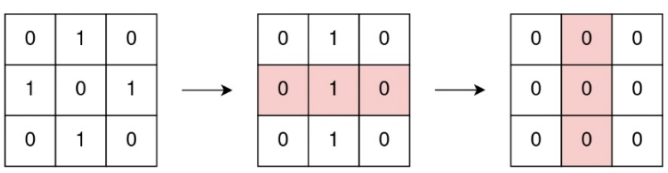
Medium | Topics | Companies | Hint

You are given an `m x n` binary matrix `grid`.

In one operation, you can choose **any** row or column and flip each value in that row or column (i.e., changing all `0`'s to `1`'s, and all `1`'s to `0`'s).

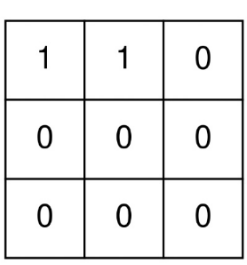
Return `true` *if it is possible to remove all* `1`*'s from* `grid` using **any** number of operations or `false` otherwise.

Example 1:



```
Input: grid = [[0,1,0],[1,0,1],[0,1,0]]
Output: true
Explanation: One possible way to remove all 1's from grid is to:
- Flip the middle row
- Flip the middle column
```

Example 2:



```
Input: grid = [[1,1,0],[0,0,0],[0,0,0]]
Output: false
Explanation: It is impossible to remove all 1's from grid.
```

Example 3:

```
Input: grid = [[0]]
Output: true
Explanation: There are no 1's in grid.
```

Constraints:

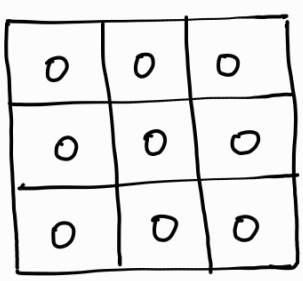
- `m == grid.length`
- `n == grid[i].length`
- `1 <= m, n <= 300`
- `grid[i][j]` is either `0` or `1`.

My understanding is:

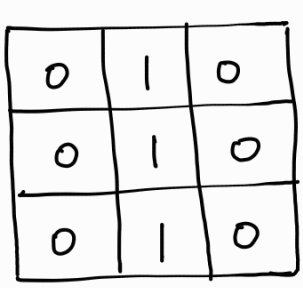
* A matrix given is a flipped version of zero matrix. If not, that matrix cannot be flipped to zero matrix and return false

o/p is either true or false.

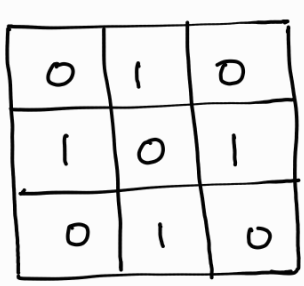
Let's confirm my understanding here:



flip col 1:



flip row 1 :



Conclusion:

See example 1 and Hint 3 in the question.

Hint 1

Does the order, in which you do the operations, matter?

Hint 2

No, it does not. An element will keep its original value if the number of operations done on it is even and vice versa. This also means that doing more than 1 operation on the same row or column is unproductive.

Hint 3

Try working backward, start with a matrix of all zeros and try to construct grid using operations.

Hint 4

Start with operations on columns, after doing them what do you notice about each row?

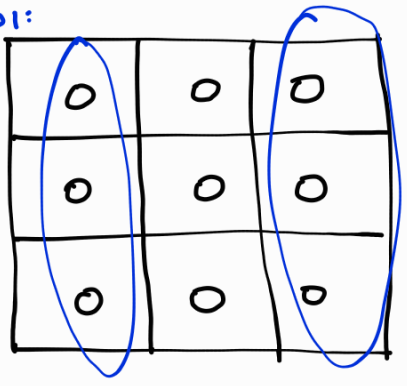
Hint 5

Each row is the exact same. If we then flip some rows, that leaves only two possible arrangements for each row: the same as the original or the opposite.

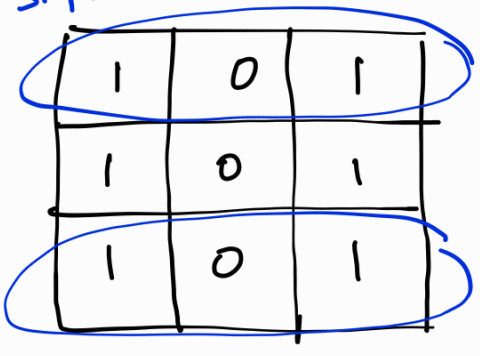
* Hint 4 & 5: If we start operations on columns and then flip rows, each row has two arrangements - original & opposite.

Let's confirm understanding:

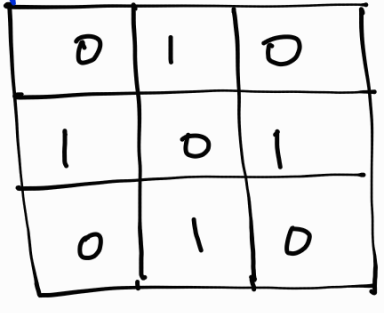
Step 1:



Step 2:

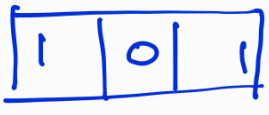


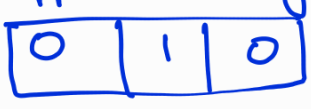
Step 3:



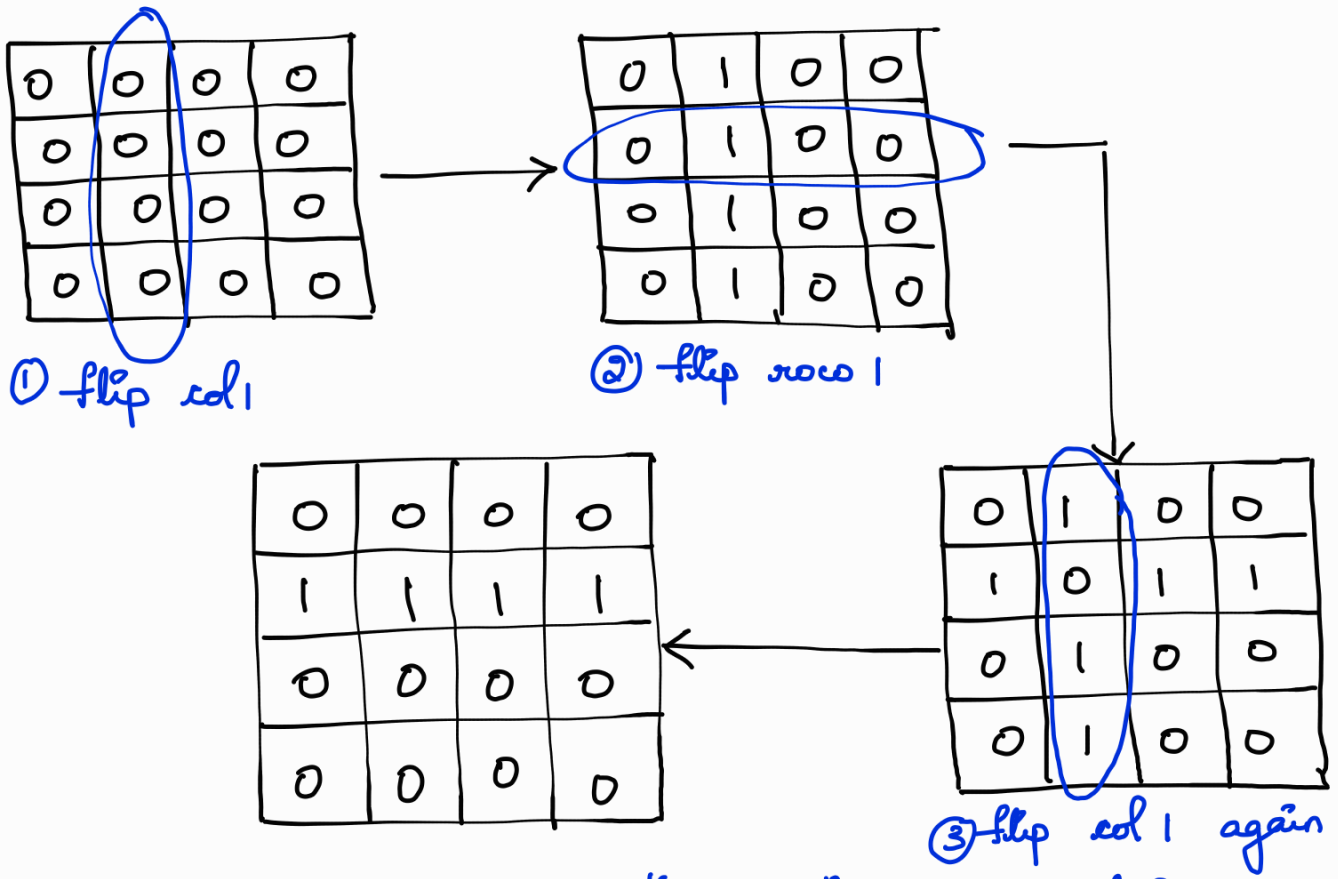
Conclusion:①

From step ② & ③, that is, after column operations first and then row operations, each row has exact same or opposite arrangement.

Step ②: 

Step ③: 

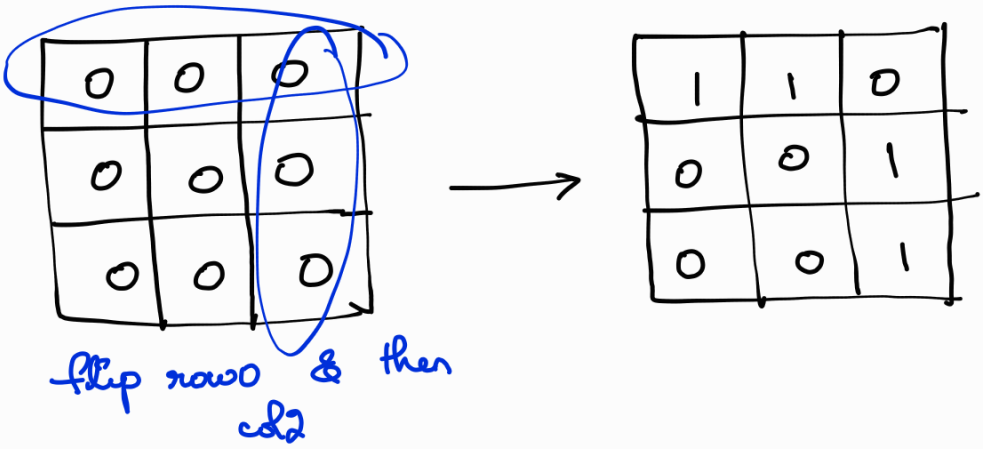
*



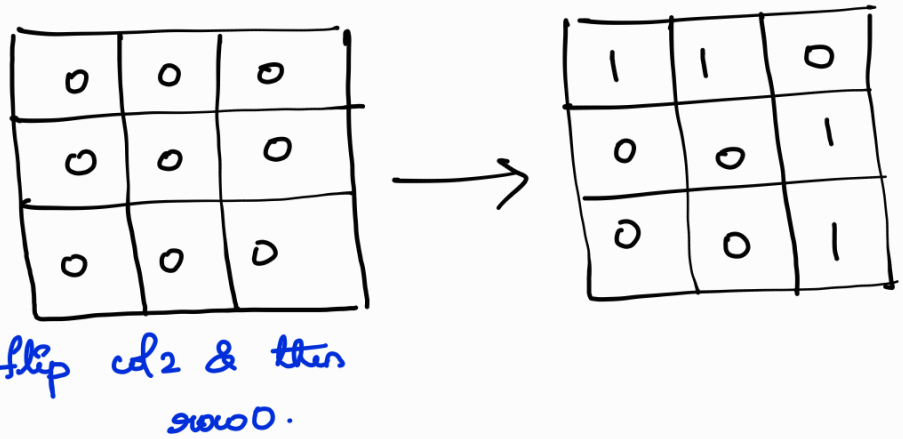
From step ① & step ③, doing "flip col 1" operation twice is equivalent to not flipping at all.

Conclusion ②: Flipping more than once means not flipping at all.

①



②



Conclusion ②: The order doesn't matter because the outcome is same.

Brute force approach :- $O(2^{cols})$

① All possible column flips :

There are n columns and each column can be either in '0' or '1' state. So there are 2^n column combinations.

Eg:- 3 cols ⇒ $2^3 = 8$ comb's (or $1 << 3$)

4 cols ⇒ $2^4 = 16$ comb's (or $1 << 4$)

n Powers of 2 can be represented as $(1 << n)$

$n = 3 \Rightarrow 1 << 3$

$\Rightarrow 0001 << 3$ (3, 2, 1)

$\Rightarrow 1000 = (8)_{10}$

② Apply Each Combination :

For each combination, apply flip to the particular column.

For a 3×3 matrix, possible column flips is :

mask ⇒

000 ⟶ indicates no flip
001 ⟶ indicates flip col 0
010 ⟶ indicates flip col 1
011 ⟶ indicates flip col 0, col 1.
100
101
110
111

Apply these combinations on columns:

For col0, col1, col2; check if these columns are 1.

col2 col1 col0

1 1 0 $= (6)_{10}$

col1 & col2 are 1.

But how do we check which columns are 1's. All we have is mask = 6.

```
for (row=0; ..... ) {
  for (col=0; col<n; col++) {   //col0, col1, col2
        // 0th column represented as 0th bit (001) -> (1<<0)
        // 1st column represented as 1st bit (010) -> (1<<1)
        // 2nd column represented as 2nd bit (100) -> (1<<2)
        :
        //(n-1)th column represented as (n-1)th bit (100.....0)
        // nth column represented as (1<<n)
        // Let's say we have a combination of
        // (101)2 = mask (=> 0th & 2nd cols are 1)
        // then we do an '&' operation of col with
        //mask to check if that 'col' is 1
        // Eg:- col=2, mask = (101)2 = 5
        //   5 & (1<<2) = 101 & 100 = 100 (!=0)
        // which is non-zero i.e., col2 should be
        // flipped
```

```
if((mask & (1<<col)) != 0){
    // flip column.
    temp[row][col] = Math.abs(
        1 - mat[row][col]);
} else {
    temp[row][col] = mat[row][col];
}
}
}
```

③ Check for valid combination :-

Compare row1 with row0

Compare row2 with row1

```
for(int i=1; i<n; i++){
    for(int j=0; j<n; j++){
        if(grid[i][j] != grid[i-1][j]){
            return false;
        }
    }
}
return true;
```

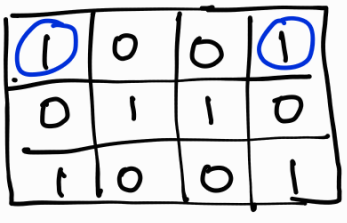
Code :-

```
public boolean removeOnes(int[][] grid) {
    int cols = grid[0].length;
    for(int mask = 0; mask < (1<<cols); mask++){
        int[][] tempGrid = flipColumns(grid, mask);
        if(checkSameSequence(tempGrid)) {
            return true;
        }
    }
    return false;
} // O(2^cols)
private int[][] flipColumns(int[][] grid, int mask) {
    int rows = grid.length;
    int cols = grid[0].length;
    int tempGrid[][] = new int[rows][cols];

    for(int i=0; i< rows; i++) {
        for(int j=0; j< cols; j++) {
            if((mask & (1<<j)) != 0) { //jth position in mask
                tempGrid[i][j] = 1-grid[i][j];

            } else {
                tempGrid[i][j] = grid[i][j];
            }
        }
    }
    return tempGrid;
} //O(rows * cols)
```

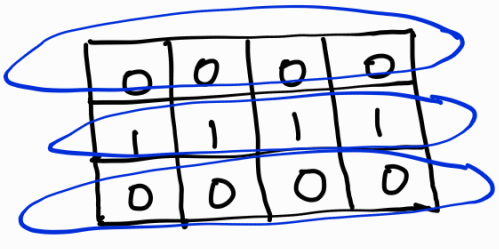
```
private boolean checkSameSequence(int[][] grid) {
    int rows = grid.length;
    int cols = grid[0].length;
    for(int i=1; i<rows; i++) {
        for(int j=0; j<cols; j++) {
            if(grid[i][j] != grid[i-1][j]) {
                return false;
            }
        }
    }
    return true;
} // O(rows * cols)
```

2) Optimised Approach :- $O(mn)$ Instead of flipping all possible combinations, we flip only necessary columns and then check row elements.

Make all 1's to 0's of first row by flipping the column.



We cannot flip row because, the 0's will become 1's. So we flip columns.



Validate if the rows have same pattern 0000 or 1111.

Code :-

```
public boolean removeOnes(int[][] grid) {
    int n = grid[0].length;
    IntStream.range(0,n).forEach(j -> {
        if (grid[0][j] == 1) flipColumn(j, grid);
    });

    for(int i=1; i<m; i++) {
        for(int j=1; j<n; j++) {
            if(grid[i][j] != grid[i][j-1]) return false;
        }
    }
    return true;
}
private void flipColumn(int j, int[][] grid) {
    IntStream.range(0, grid.length).forEach(i -> {
        grid[i][j] = 1 - grid[i][j];
    });
}
```

Note: In this approach, instead of checking all possible columns combinations

Basically all the row patterns should be same.

11011 or 00100 are same. Then the matrix can be made zero matrix. (refer Conclusion ①)

Simple Code :- O(mn)

```
removeOnes (int[][] grid) {
    for (int i=1; i< grid.length; i++) {
        boolean same = true;
        boolean opposite = true;
        for (int j=0; j< grid[0].length; j++) {
            same = same && grid[0][j] == grid[i][j];
            opposite = opposite && grid[0][j] != grid[i][j];
        }
        if (!same && !opposite) {
            return false;
        }
    }
    return true;
}
```

6)

2174. Remove All Ones With Row and Column Flips II Premium

Medium | Topics | Companies | Hint

You are given a **0-indexed** `m x n` **binary** matrix `grid`.

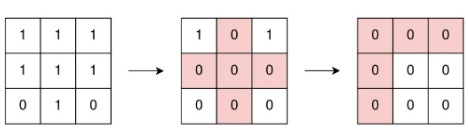
In one operation, you can choose any `i` and `j` that meet the following conditions:

- `0 <= i < m`
- `0 <= j < n`
- `grid[i][j] == 1`

and change the values of **all** cells in row `i` and column `j` to zero.

Return *the* ***minimum*** *number of operations needed to remove all* `1`*'s from* `grid`.

Example 1:



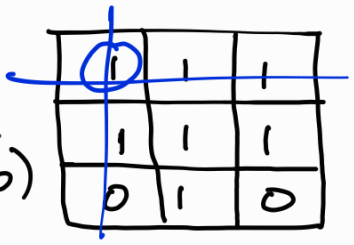
Input: grid = [[1,1,1],[1,1,1],[0,1,0]]
Output: 2
Explanation:
In the first operation, change all cell values of row 1 and column 1 to zero.
In the second operation, change all cell values of row 0 and column 0 to zero.

Brute force solution :- (DFS & Backtracking)

Explore each cell and if it's 1, then take backup of that row (rowValues) and that column (columnValues). Then set that row and column to zeroes. Now, count the current flip and reset the matrix, then repeat process for next cell.

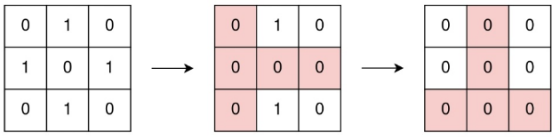
At each process, update minimum flips.

Explanation :-



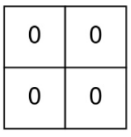
Process (1): (row 0, col 0) = 1

Example 2:



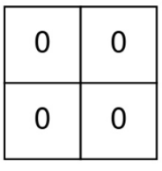
```
Input: grid = [[0,1,0],[1,0,1],[0,1,0]]
Output: 2
Explanation:
In the first operation, change all cell values of row
1 and column 0 to zero.
In the second operation, change all cell values of
row 2 and column 1 to zero.
Note that we cannot perform an operation using row 1
and column 1 because grid[1][1] != 1.
```

Example 3:



```
Input: grid = [[0,0],[0,0]]
Output: 0
Explanation:
There are no 1's to remove so return 0.
```

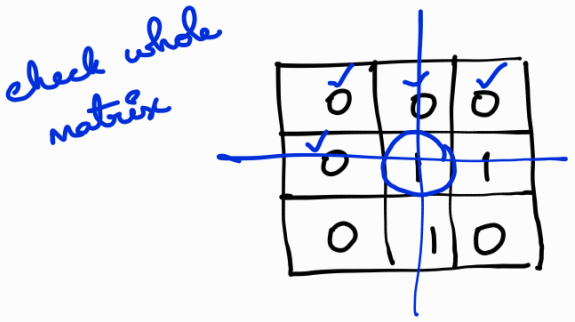
Example 3:

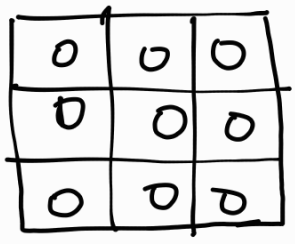


```
Input: grid = [[0,0],[0,0]]
Output: 0
Explanation:
There are no 1's to remove so return 0.
```

Constraints:

- `m == grid.length`
- `n == grid[i].length`
- `1 <= m, n <= 15`
- `1 <= m * n <= 15`
- `grid[i][j]` is either `0` or `1`.



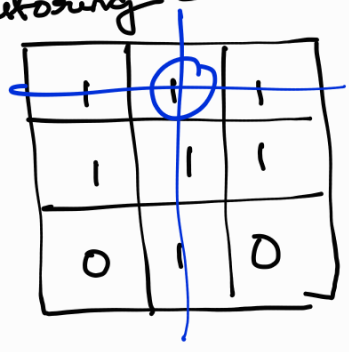


flips = 2

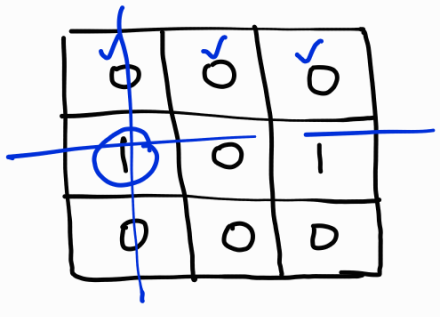
minflips = 2

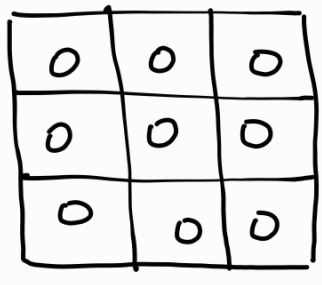
Process ②: restoring back

(row2, col1) = 1



check whole matrix.

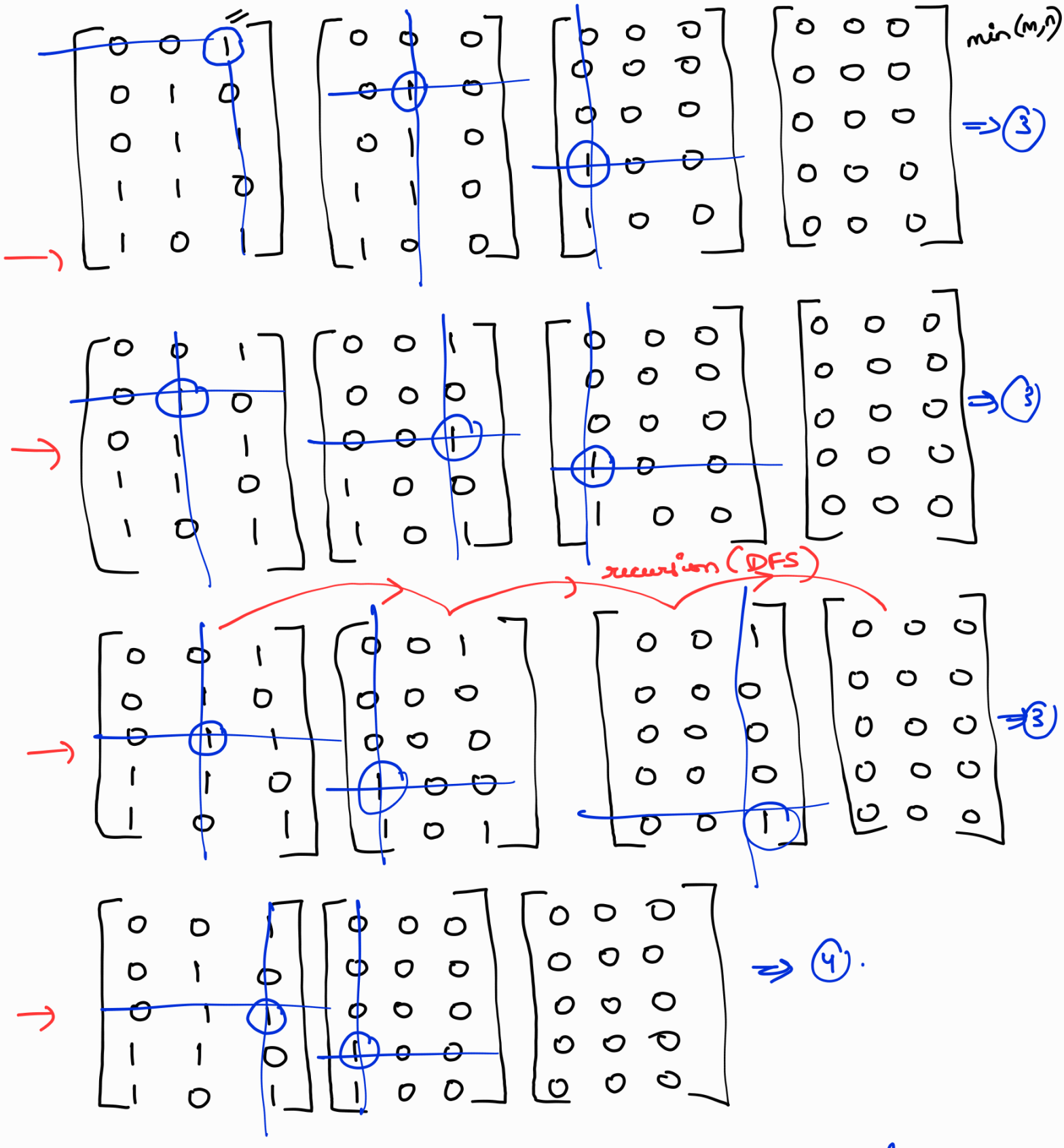




flips = 2

minflips = 2

This goes on. As we observe, this is how we manually check and hence we will implement code accordingly.



This approach of DFS and reset the data to Original is Backtracking.

Code (Backtracking) :-

```java
class Solution {
    public int removeOnes(int[][] grid) {
        int M = grid.length, N = grid[0].length, minFlips = Integer.MAX_VALUE;
        int[] rowValues = new int[N], colValues = new int[M];
        for(int row = 0; row < M; row++) {
            for(int col = 0; col < N; col++) {
                if(grid[row][col] == 0) {
                    continue;
                }
                for(int r = 0; r < M; r++) {
                    colValues[r] = grid[r][col];
                    grid[r][col] = 0;
                }
                for(int c = 0; c < N; c++) {
                    rowValues[c] = grid[row][c];
                    grid[row][c] = 0;
                }
                minFlips = Math.min(minFlips, (1 + removeOnes(grid)));
                for(int c = 0; c < N; c++) {
                    grid[row][c] = rowValues[c];
                }
                for(int r = 0; r < M; r++) {
                    grid[r][col] = colValues[r];
                }
            }
        }
        return ((minFlips == Integer.MAX_VALUE) ? 0 : minFlips);
    }
}
```

restore row & col.

Time Complexity :-

(1) There are $m * n$ iterations

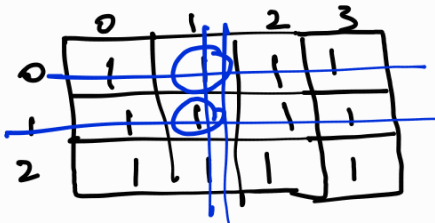
itr 0 itr 1 itr 2 itr 3 ..... itr $m*n$

(2) At each iteration, there is a recursion call.

So complexity overall = mn (iterations) * [complexity of recursion]

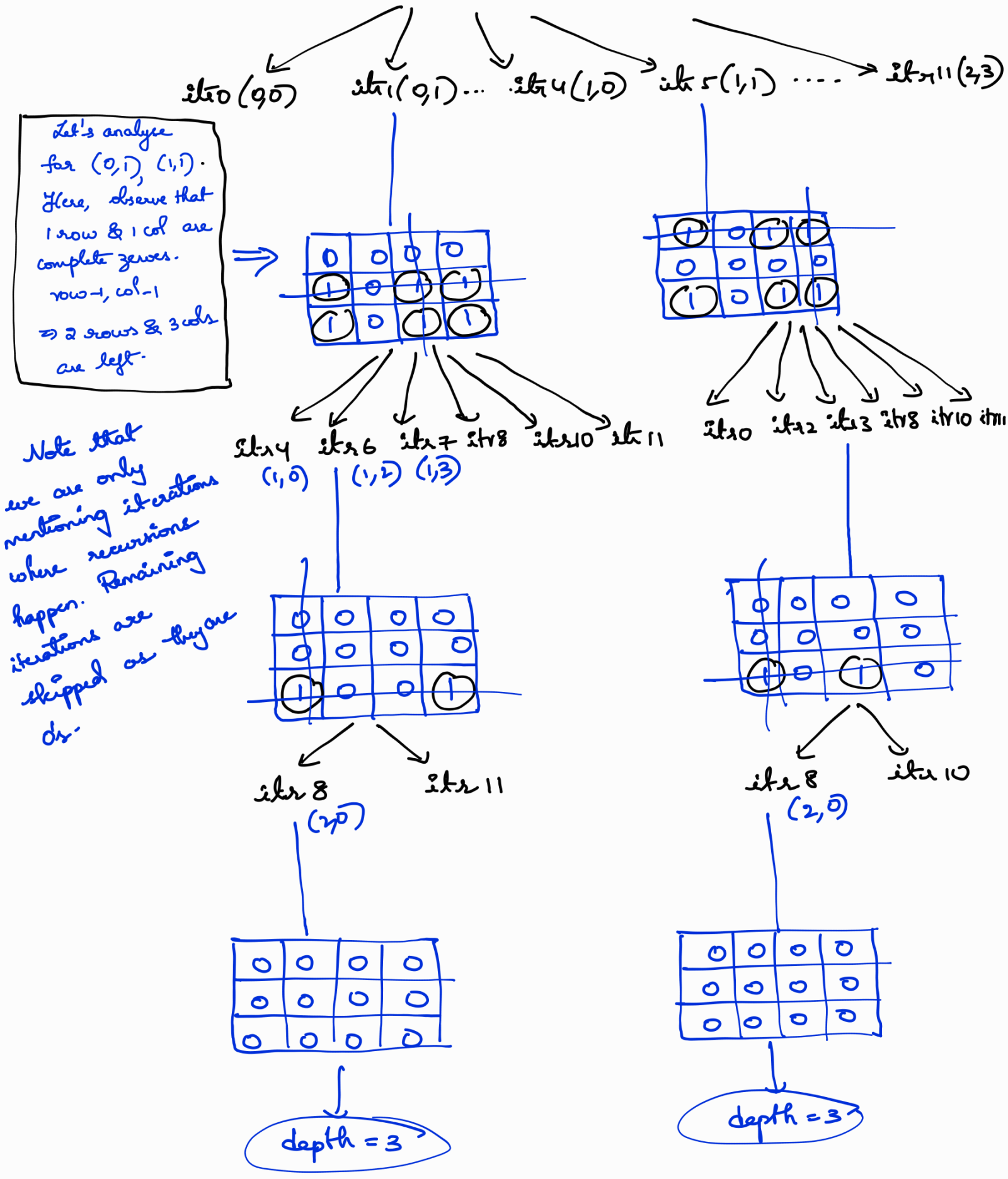
(3) Let's analyse max recursion depth of each recursion.

Ex:



$m = 3$

$n = 4$.



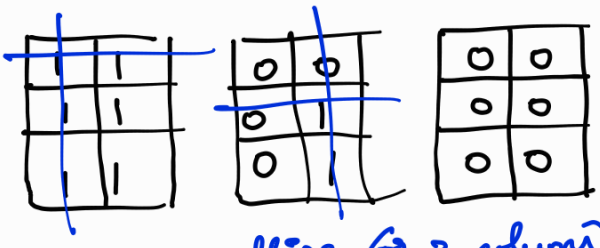
Note that max-depth of recursion tree is 3 and at each level we have (mn) iterations.

The max-depth of recursion tree can be written as min(m,n)
A matrix can be made into full zero in min(m,n) flips.

Each flip involves updating the row and the column of a cell to zeroes. For a m×n matrix, we need min(m,n) flips which is what we are achieving per recursion depth.

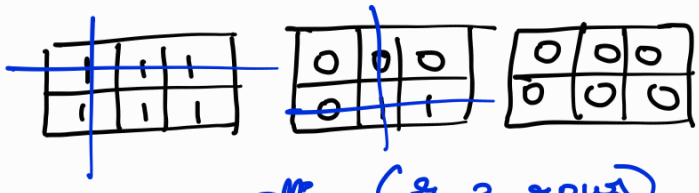
See the below example to understand.

3×2 matrix



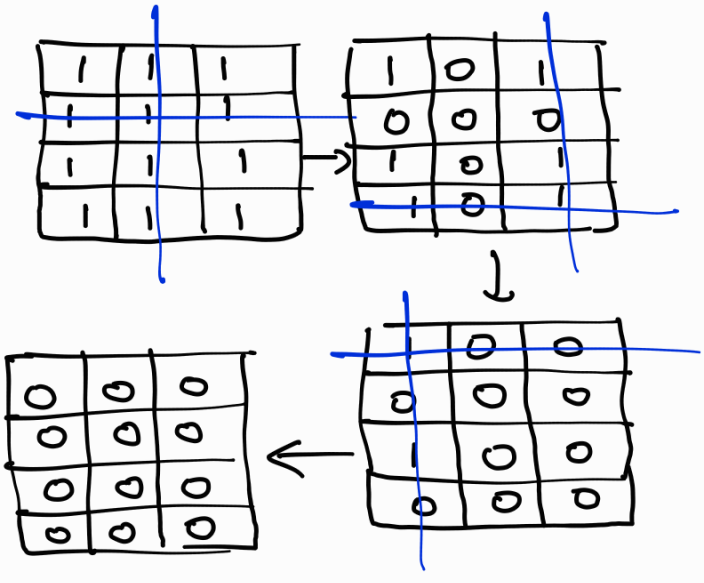
2 flips (& 2 columns)

2×3 matrix



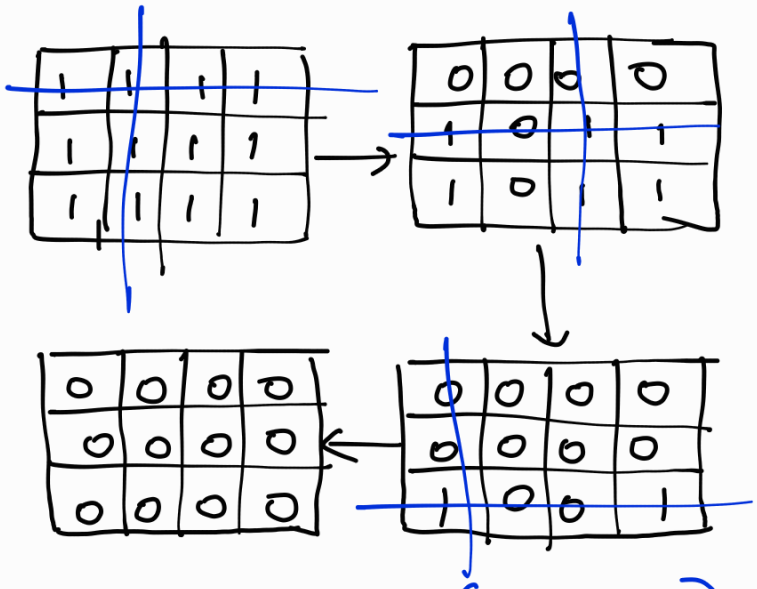
2 flips (& 2 rows)

4×3 matrix



3 flips (& 3 columns)

3×4 matrix

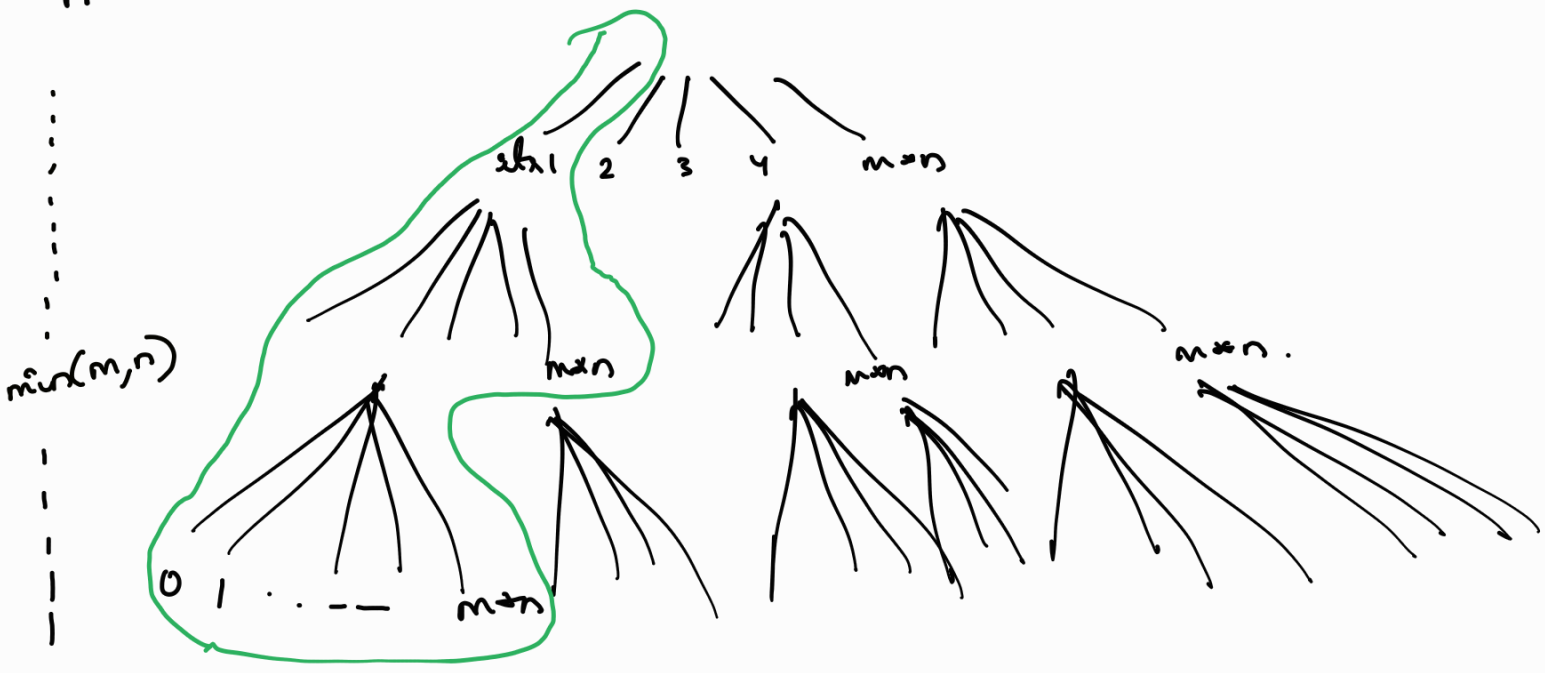


3 flips (& 3 rows)

If we consider any random flips on a matrix, the zero matrix is obtained in min(m,n) flips.

There are (m*n) such flips as there are (m*n) 1's (in worst case).

④ In point ③, at each level of recursion (n*n) iterations happen and each iteration has recursion.



∴ Per recursion depth, operations happen as follows:

$(mn) * (mn) * (mn) \ldots\ldots$ $\min(m,n)$ times.

Because, per iteration in mn iterations, there is again mn iterations, and continues $\min(m,n)$ times.

∴ The time complexity is: $(mn)^{\min(m,n)}$.

BFS approach:-

(BFS approach + bitmasking)

```java
class Solution {
    public int removeOnes(int[][] grid) {
        int m = grid.length, n = grid[0].length;
        long state = 0;
        // Convert grid to bit, the bit is 1 if grid[i][j] is 1
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                int k = i * n + j;
                state |= ((long)(1 << k)) * grid[i][j];
            }
        }
        Queue<Long> q = new LinkedList<>();
        q.add(state);
        int step = 0;
        HashSet<Long> seen = new HashSet<>();
        seen.add(state);
        while (q.size() > 0) {
            int size = q.size();
            while (size -- > 0) {
                long cur = q.remove();
                if (cur == 0) return step;
                ArrayList<Integer> lst = new ArrayList<>();
                for (int i = 0; i < m; ++i) {
                    for (int j = 0; j < n; ++j) {
                        int k = i * n + j;
                        if ((cur & (1 << k)) > 0) {
                            lst.add(k);
                        }
                    }
                }
                for (Integer e : lst) {
                    int r = e / n, c = e % n;
                    long nstate = cur;
                    for (int i =0; i < n; ++i) {
                        int k = r * n + i;
                        nstate &= (~(1 << k));
                    }
                    for (int i =0; i < m; ++i) {
                        int k = i * n + c;
                        nstate &= (~(1 << k));
                    }
                    if (!seen.contains(nstate)) {
                        q.add(nstate);
                        seen.add(nstate);
                    }
                }
            }
            step ++;
        }
        return -1;
    }
}
```

convert matrix to a bit representation.

$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = (1001)_2$

add the state to queue.

add it to the seen.

add positions of 1s to list

set the position of k^{th} bit to 0.

add new state to the queue and seen.

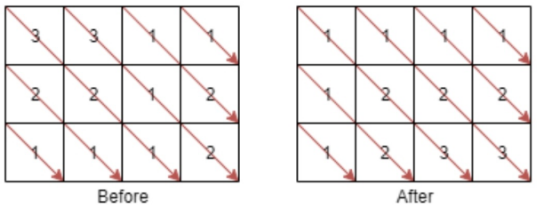
1329. Sort the Matrix Diagonally

Medium | Topics | Companies | Hint

A **matrix diagonal** is a diagonal line of cells starting from some cell in either the topmost row or leftmost column and going in the bottom-right direction until reaching the matrix's end. For example, the **matrix diagonal** starting from `mat[2][0]`, where `mat` is a `6 x 3` matrix, includes cells `mat[2][0]`, `mat[3][1]`, and `mat[4][2]`.

Given an `m x n` matrix `mat` of integers, sort each **matrix diagonal** in ascending order and return *the resulting matrix*.

Example 1:




```
Input: mat = [[3,3,1,1],[2,2,1,2],[1,1,1,2]]
Output: [[1,1,1,1],[1,2,2,2],[1,2,3,3]]
```

Example 2:

```
Input: mat = [[11,25,66,1,69,7],[23,55,17,45,15,52],
[75,31,36,44,58,8],[22,27,33,25,68,4],[84,28,14,11,5,50]]
Output: [[5,17,4,1,52,7],[11,11,25,45,8,69],
[14,23,25,44,58,15],[22,27,31,36,50,66],[84,28,75,33,55,68]]
```

Constraints:

- `m == mat.length`
- `n == mat[i].length`
- `1 <= m, n <= 100`
- `1 <= mat[i][j] <= 100`

Each diagonal has elements whose (i-j) is same.

Hence we add diagonal elements to hash table mapped by (i-j) key.

The value of the hash table is a Heap datastructure that holds ordered data.

Code :-

```
HashMap<Integer, PriorityQueue<Integer>>
 hm = new HashMap<>();
for(int i=0; i<m; i++) {
  for(int j=0; j<n; j++) {
    hm.putIfAbsent(i-j,
                   new PriorityQueue<>());
    hm.get(i-j).add(mat[i][j]);
  }
}
for(int i=0; i<m; i++) {
  for(int j=0; j<n; j++) {
    mat[i][j] = hm.get(i-j).poll();
  }
}
```

Time Complexity :- O(mn).

Another similar code using Heap:

```
class Solution {

    PriorityQueue<Integer> pQ = new PriorityQueue<>();

    public int[][] diagonalSort(int[][] mat) {

        int[][] seen = new int[mat.length][mat[0].length];

        for(int i = 0; i < mat.length; i++) {
            for(int j = 0; j < mat[0].length; j++) {
                if(seen[i][j] != 1) {
                    loadQueue(mat, i, j);
                    loadDiagonal(mat, seen, i, j);
                }
            }
        }

        return mat;
    }


    public void loadQueue(int[][] mat, int i, int j) {
        if(i >= mat.length || j >= mat[0].length || i < 0 || j < 0) {
            return;
        }

        pQ.add(mat[i][j]);
        loadQueue(mat, i + 1, j + 1);
    }

    public void loadDiagonal(int[][] mat, int[][] seen,  int i, int j) {
        if(i >= mat.length || j > mat[0].length || i < 0 || j < 0 || pQ.isEmpty())
            return;
        }

        mat[i][j] = pQ.poll();
        seen[i][j] = 1;
        loadDiagonal(mat, seen, i + 1, j + 1);
    }
}
```