

Q28: Transpose of Matrix

1. Understand the Problem

- **Read & Identify:** We are given a square matrix of size $N \times N$. We need to transpose it, i.e., switch rows and columns.
 - **Goal:** Convert $\text{matrix}[i][j]$ to $\text{matrix}[j][i]$. Do it in-place without using extra space.
 - **Paraphrase:** Swap elements above the diagonal with the corresponding elements below the diagonal.
-

2. Constraints

- $1 \leq N \leq 100$
 - $-10^3 \leq \text{mat}[i][j] \leq 10^3$
-

3. Examples & Edge Cases

Example 1 (Normal Case): Input:

```
4
1 1 1 1
2 2 2 2
3 3 3 3
4 4 4 4
```

Output:

```
1 2 3 4
1 2 3 4
1 2 3 4
1 2 3 4
```

4. Approaches

Approach 1: Brute Force (Using Extra Matrix)

- **Idea:**
 - Create a new matrix $\text{transpose}[N][N]$
 - Set $\text{transpose}[j][i] = \text{matrix}[i][j]$

- Print the new matrix

Java Code:

```
public static void transposeMatrixBrute(int[][] mat) {
    int N = mat.length;
    int[][] trans = new int[N][N];
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            trans[j][i] = mat[i][j];
        }
    }

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            System.out.print(trans[i][j] + " ");
        }
        System.out.println();
    }
}
```

Complexity:

- Time: $O(N^2)$
- Space: $O(N^2)$

Approach 2: Optimal (In-Place Swap)

- **Idea:**
 - Only swap elements above the diagonal with elements below the diagonal
 - For all $i < j$, swap $\text{mat}[i][j]$ with $\text{mat}[j][i]$

Java Code:

```
public static void transposeMatrixOptimal(int[][] mat) {
    int N = mat.length;

    for (int i = 0; i < N; i++) {
        for (int j = i + 1; j < N; j++) {
            int temp = mat[i][j];
            mat[i][j] = mat[j][i];
            mat[j][i] = temp;
        }
    }

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            System.out.print(mat[i][j] + " ");
        }
        System.out.println();
    }
}
```

```
    }  
}
```

Complexity:

- Time: $O(N^2)$ â†’ every element visited once
 - Space: $O(1)$ â†’ in-place
-

5. Justification / Proof of Optimality

- Brute force works but uses extra space
 - Optimal approach modifies the matrix in-place â†’ meets the expected space complexity
 - Swapping above diagonal with below diagonal ensures all elements are transposed
-

6. Variants / Follow-Ups

- Transpose non-square matrix â†’ requires extra matrix
- Rotate matrix 90° , 180° â†’ transpose + reverse operations
- Compute transpose without changing original matrix
- Use in algorithms like matrix multiplication optimization

Q29: Rotate a Matrix by 90° (Clockwise & Anti-Clockwise)

1. Understand the Problem

- **Read & Identify:** We are given an $n \times n$ square matrix. We need to rotate it 90° clockwise or 90° anti-clockwise.
 - **Goal:** Clockwise: top-left â†’ top-right â†’ bottom-right â†’ bottom-left Anti-clockwise: top-left â†’ bottom-left â†’ bottom-right â†’ top-right
 - **Paraphrase:** Rotate the matrix in-place using minimal extra space, ideally $O(1)$.
-

2. Constraints

- $1 \leq n \leq 100$
 - In-place solution required
-

3. Examples & Edge Cases

Example 1 (Clockwise 90° Rotation): Input:

```
3 3
7 2 3
2 3 4
5 6 1
```

Output:

```
5 2 7
6 3 2
1 4 3
```

Example 2 (Anti-Clockwise 90° Rotation): Input:

```
3 3
7 2 3
2 3 4
5 6 1
```

Output:

```
3 4 1
2 3 6
7 2 5
```

4. Approaches

Approach 1: Brute Force (Using Extra Matrix)

- **Idea:**
 - Clockwise: $\text{rotated}[j][n-1-i] = \text{matrix}[i][j]$
 - Anti-clockwise: $\text{rotated}[n-1-j][i] = \text{matrix}[i][j]$

Java Code:

```
// Clockwise
public static int[][] rotateMatrixClockwiseBrute(int[][] mat) {
    int n = mat.length;
    int[][] rotated = new int[n][n];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
```

```

        rotated[j][n-1-i] = mat[i][j];
    }
}
return rotated;
}

// Anti-Clockwise
public static int[][] rotateMatrixAntiClockwiseBrute(int[][] mat) {
    int n = mat.length;
    int[][] rotated = new int[n][n];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            rotated[n-1-j][i] = mat[i][j];
        }
    }
    return rotated;
}

```

Complexity:

- Time: $O(N^2)$
- Space: $O(n^2)$ â†' extra matrix

Approach 2: Optimal In-Place (Transpose + Reverse)

- **Idea:**
 - Idea (Clockwise):
 - Transpose matrix in-place
 - Reverse each row
 - Idea (Anti-Clockwise):
 - Transpose matrix in-place
 - Reverse each column

Java Code:

```

// Clockwise 90°
public static void rotateMatrixClockwise(int[][] mat) {
    int n = mat.length;
    // Transpose
    for (int i = 0; i < n; i++) {
        for (int j = i+1; j < n; j++) {
            int temp = mat[i][j];
            mat[i][j] = mat[j][i];
            mat[j][i] = temp;
        }
    }
    // Reverse rows
    for (int i = 0; i < n; i++) {
        int left = 0, right = n-1;
        while (left < right) {
            int temp = mat[i][left];
            mat[i][left] = mat[i][right];
            mat[i][right] = temp;
            left++;
            right--;
        }
    }
}

```

```

        mat[i][left] = mat[i][right];
        mat[i][right] = temp;
        left++; right--;
    }
}
}

// Anti-Clockwise 90°
public static void rotateMatrixAntiClockwise(int[][] mat) {
    int n = mat.length;
    // Transpose
    for (int i = 0; i < n; i++) {
        for (int j = i+1; j < n; j++) {
            int temp = mat[i][j];
            mat[i][j] = mat[j][i];
            mat[j][i] = temp;
        }
    }
    // Reverse columns
    for (int j = 0; j < n; j++) {
        int top = 0, bottom = n-1;
        while (top < bottom) {
            int temp = mat[top][j];
            mat[top][j] = mat[bottom][j];
            mat[bottom][j] = temp;
            top++; bottom--;
        }
    }
}

```

Complexity:

- Time: $O(n^2)$ â†’ transpose + reverse
 - Space: $O(1)$ â†’ in-place
-

5. Justification / Proof of Optimality

- Brute force is simple but uses extra space
 - Optimal approach rotates in-place â†’ meets constraints
 - Works for any square matrix, handles maximum n efficiently
-

6. Variants / Follow-Ups

- Rotate 180° â†’ two 90° rotations
- Rotate by arbitrary angle (multiple of 90°)
- Rotate non-square matrix â†’ requires extra matrix
- Combine rotation with matrix reflection operations

Q30: Find The Way (Mouse in Binary Matrix)

1. Understand the Problem

- **Read & Identify:** A mouse enters a binary matrix at (0,0) moving left to right. It moves straight on 0. It turns right and changes 1 → 0 when it encounters 1. Determine exit coordinates when the mouse leaves the matrix.
 - **Goal:** Simulate the mouse movement until it exits the matrix.
 - **Paraphrase:** Start at (0,0), follow direction rules: 0 → continue 1 → turn right, set 1 → 0 Stop when the next move goes outside the matrix bounds
-

2. Constraints

- $1 \leq m, n \leq 100$
 - $\text{matrix}[i][j] \in \{0,1\}$
-

3. Examples & Edge Cases

Example 1 (Normal Case): Input:

```
3 3
0 1 0
0 1 0
1 0 1
```

Output:

```
1 0
```

Example 2 (Normal Case): Input:

```
1 2
0 0
```

Output:

0 1

4. Approaches

Approach 1: Brute Force (Simulate Movement)

- **Idea:**

- Maintain current position (i, j) and direction (0:right,1:down,2:left,3:up)
- Move according to rules:
- $\text{matrix}[i][j] = 0 \rightarrow$ continue
- $\text{matrix}[i][j] = 1 \rightarrow$ turn right, set to 0
- Stop when (i,j) goes outside bounds

Java Code:

```
public static int[] findMouseExit(int[][] mat) {  
    int m = mat.length;  
    int n = mat[0].length;  
  
    int dir = 0; // 0=right, 1=down, 2=left, 3=up  
    int i = 0, j = 0;  
  
    while (i >= 0 && i < m && j >= 0 && j < n) {  
        if (mat[i][j] == 1) {  
            mat[i][j] = 0; // change 1 to 0  
            dir = (dir + 1) % 4; // turn right  
        }  
  
        // Move in current direction  
        if (dir == 0) j++;  
        else if (dir == 1) i++;  
        else if (dir == 2) j--;  
        else if (dir == 3) i--;  
    }  
  
    // Exit is last valid position  
    if (dir == 0) j--; // right exit  
    else if (dir == 1) i--; // down exit  
    else if (dir == 2) j++; // left exit  
    else if (dir == 3) i++; // up exit  
  
    return new int[]{i, j};  
}
```

Complexity:

- Time: $O(m*n)$ since each cell is visited at most once
- Space: $O(1)$ in-place

5. Justification / Proof of Optimality

- Simulating step-by-step ensures all turns and updates are handled correctly.
 - Changing 1 → 0 prevents infinite loops.
 - Works for all matrix sizes within constraints.
-

6. Variants / Follow-Ups

- Mouse starting at arbitrary cell
- Multiple mice moving simultaneously → detect collisions
- 3D matrix → simulate 3D movement rules
- Count steps until exit

Q31: Spirally Traversing a Matrix (Clockwise & Anti-clockwise)

1. Understand the Problem

- **Read & Identify:** You are given a matrix of size $m \times n$. Task is to print all elements in spiral order:
Clockwise → top row → right column → bottom row → left column → shrink boundaries, repeat.
Anti-clockwise → left column → bottom row → right column → top row → shrink boundaries, repeat.
 - **Goal:** Print traversal order of the matrix elements.
 - **Paraphrase:** Keep four boundaries (top, bottom, left, right) and traverse them layer by layer until all elements are covered.
-

2. Constraints

- $1 \leq m, n \leq 100$
 - $-10^3 \leq \text{mat}[i][j] \leq 10^3$
-

3. Examples & Edge Cases

Example 1 (Clockwise): Input:

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

Output:

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

Example 2 (Anti-clockwise): Input:

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

Output:

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

4. Approaches

Approach 1: Brute Force (Simulate Layer-by-Layer)

- **Idea:**
 - Maintain top, bottom, left, right indices.
 - Clockwise order: top → right → bottom → left.
 - Anti-clockwise order: left → bottom → right → top.
 - After traversing a boundary, shrink it (top++, bottom--, left++, right--).

Java Code:

Java Code (Clockwise Spiral Traversal)

```
public static void spiralClockwise(int[][] mat) {
    int m = mat.length, n = mat[0].length;
    int top = 0, bottom = m - 1, left = 0, right = n - 1;

    while (top <= bottom && left <= right) {
        // Traverse top row
        for (int j = left; j <= right; j++)
            System.out.print(mat[top][j] + " ");
        top++;

        // Traverse right column
        for (int i = top; i <= bottom; i++)
            System.out.print(mat[i][right] + " ");
        right--;
    }
}
```

```

        // Traverse bottom row
        if (top <= bottom) {
            for (int j = right; j >= left; j--)
                System.out.print(mat[bottom][j] + " ");
            bottom--;
        }

        // Traverse left column
        if (left <= right) {
            for (int i = bottom; i >= top; i--)
                System.out.print(mat[i][left] + " ");
            left++;
        }
    }
}

```

Java Code (Anti-clockwise Spiral Traversal)

```

public static void spiralAntiClockwise(int[][] mat) {
    int m = mat.length, n = mat[0].length;
    int top = 0, bottom = m - 1, left = 0, right = n - 1;

    while (top <= bottom && left <= right) {
        // Traverse left column
        for (int i = top; i <= bottom; i++)
            System.out.print(mat[i][left] + " ");
        left++;

        // Traverse bottom row
        for (int j = left; j <= right; j++)
            System.out.print(mat[bottom][j] + " ");
        bottom--;

        // Traverse right column
        if (left <= right) {
            for (int i = bottom; i >= top; i--)
                System.out.print(mat[i][right] + " ");
            right--;
        }

        // Traverse top row
        if (top <= bottom) {
            for (int j = right; j >= left; j--)
                System.out.print(mat[top][j] + " ");
            top++;
        }
    }
}

```

Complexity:

- Time: $O(m \cdot n)$ Every element is visited once.
 - Space: $O(1)$ in-place
-

5. Variants / Follow-Ups

- Return spiral traversal as an array instead of printing.
- Zigzag spiral traversal.
- Multi-layered spiral (spiral inward, then outward).

Q32: Sum of Upper and Lower Triangles (Primary & Secondary Diagonals)

1. Understand the Problem

- **Read & Identify:** Given an $n \times n$ square matrix. Task: Calculate sum of upper triangle and lower triangle w.r.t a diagonal. Upper Triangle: diagonal + elements above it. Lower Triangle: diagonal + elements below it. Can be asked for: Primary diagonal (\sim top-left \rightarrow bottom-right). Secondary diagonal (\nwarrow top-right \rightarrow bottom-left).
-

2. Constraints

- $1 \leq n \leq 1000$
 - $-10^3 \leq \text{mat}[i][j] \leq 10^3$
-

3. Examples & Edge Cases

Example 1 (Primary Diagonal): Input:

```
3
1 2 3
1 5 3
4 5 6
```

Output:

```
20 22
Explanation:
```

Upper (primary): $1 + 2 + 3 + 5 + 3 + 6 = 20$

Lower (primary): $1 + 1 + 4 + 5 + 5 + 6 = 22$

Example 2 (Secondary Diagona): Input:

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

Output:

23 21

Explanation:

Upper (secondary): includes diagonal + above it $3 + 2 + 1 + 5 + 7 + 9 = 23$

Lower (secondary): includes diagonal + below it $3 + 6 + 9 + 5 + 8 = 21$

4. Approaches

Approach 1: Brute Force

- **Idea:**

- Traverse entire matrix twice:
- First time \rightarrow compute upper triangle sum.
- Second time \rightarrow compute lower triangle sum.
- Works but has unnecessary double traversal.

Java Code:

```
public static void triangleSumsBrute(int n, int[][] mat, boolean primary) {
    int upper = 0, lower = 0;

    // Upper Triangle
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (primary) { // primary diagonal
                if (j >= i) upper += mat[i][j];
            } else { // secondary diagonal
                if (i + j <= n - 1) upper += mat[i][j];
            }
        }
    }
}
```

```

// Lower Triangle
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        if (primary) {
            if (j <= i) lower += mat[i][j];
        } else {
            if (i + j >= n - 1) lower += mat[i][j];
        }
    }
}

System.out.println(upper + " " + lower);
}

```

Complexity:

- Time: $O(n^2) - 2 = O(n^2)$.
- Space: $O(1)$.

Approach 2: Optimal (Single Traversal)

- **Idea:**
 - Traverse matrix once.
 - Use conditions:
 - Primary Diagonal:
 - $j \geq i \rightarrow$ upper, $j \leq i \rightarrow$ lower.
 - Secondary Diagonal:
 - $i + j \leq n-1 \rightarrow$ upper, $i + j \geq n-1 \rightarrow$ lower.

Java Code:

```

public static void triangleSumsOptimal(int n, int[][] mat, boolean primary) {
    int upper = 0, lower = 0;

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (primary) { // Primary Diagonal
                if (j >= i) upper += mat[i][j];
                if (j <= i) lower += mat[i][j];
            } else { // Secondary Diagonal
                if (i + j <= n - 1) upper += mat[i][j];
                if (i + j >= n - 1) lower += mat[i][j];
            }
        }
    }

    System.out.println(upper + " " + lower);
}

```

Complexity:

- Time: $O(n^2)$.
 - Space: $O(1)$.
-

5. Justification / Proof of Optimality

- Brute Force
 - Traverse twice â†' once for upper, once for lower.
 - Simple to understand, easy to debug.
 - Works for both primary & secondary diagonals by just changing the condition.
 - But redundant (two passes).
 - Optimal (Single Traversal)
 - Compute both sums in one loop.
 - Reduces traversal overhead â†' still $O(n^2)$ but constant factors smaller.
 - Cleaner & more efficient in practice.
-

6. Variants / Follow-Ups

- Variant A: Primary Diagonal (with diagonal) âœ... (default problem).
- Variant B: Secondary Diagonal (with diagonal) âœ... (extension).
- Variant C: Both diagonals at once âœ... (extended version).
- Variant D: Excluding diagonals âœ... (less common, but sometimes asked).

Q33: Toeplitz Matrix (Primary & Secondary)

1. Understand the Problem

- **Read & Identify:** We are given an $m \times n$ matrix. Primary Toeplitz: Every \hat{t}^{\sim} diagonal (from top-left to bottom-right) has the same value. Secondary Toeplitz: Every \hat{t}^{\sim} diagonal (from top-right to bottom-left) has the same value.
 - **Goal:** Return true if the matrix is Toeplitz in the chosen sense (Primary or Secondary)
 - **Paraphrase:** Primary: Check if $\text{mat}[i][j] == \text{mat}[i-1][j-1]$. Secondary: Check if $\text{mat}[i][j] == \text{mat}[i-1][j+1]$.
-

2. Constraints

- $1 \leq m, n \leq 20$
 - $-1000 \leq \text{matrix}[i][j] \leq 1000$
-

3. Examples & Edge Cases

Example 1 (Primary Diagonal): Input:

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

Output:

```
true
```

Explanation:

All \hat{a}^{\sim} diagonals (Primary) are equal \hat{a}^{\prime} 1,1,1, 2,2, 3, etc.

Example 2 (Secondary Diagona): Input:

```
3 3
1 2 3
2 3 1
3 1 2
```

Output:

```
true
```

Explanation:

All \hat{a}^{\sim} diagonals (Secondary) are equal \hat{a}^{\prime} 3,3,3, 2,2, 1, etc.

4. Approaches

Approach 1: Brute Force (Check Both Separately)

- **Idea:**
 - Use two boolean flags:
 - Check Primary Toeplitz ($\text{mat}[i][j] == \text{mat}[i-1][j-1]$).
 - Check Secondary Toeplitz ($\text{mat}[i][j] == \text{mat}[i-1][j+1]$).

Java Code:

```
class Solution {
    public String checkToeplitz(int[][] matrix) {
        int m = matrix.length, n = matrix[0].length;
```

```

boolean isPrimary = true;
boolean isSecondary = true;

for (int i = 1; i < m; i++) {
    for (int j = 0; j < n; j++) {
        // Primary Toeplitz check:  $i-1, j-1$ 
        if (j > 0 && matrix[i][j] != matrix[i - 1][j - 1]) {
            isPrimary = false;
        }
        // Secondary Toeplitz check:  $i-1, j+1$ 
        if (j < n - 1 && matrix[i][j] != matrix[i - 1][j + 1]) {
            isSecondary = false;
        }
    }
}

if (isPrimary && isSecondary) return "Both";
if (isPrimary) return "Primary Toeplitz";
if (isSecondary) return "Secondary Toeplitz";
return "None";
}
}

```

Complexity:

- Time: $O(m * n)$
- Space: $O(1)$

Approach 2: HashMap (Unified Indexing)

- **Idea:**
 - Use keys to group diagonals:
 - Primary key = $i - j$.
 - Secondary key = $i + j$.
 - Store the first element for each diagonal in a map and check others.

Java Code:

```

import java.util.*;

class Solution {
    public String checkToeplitz(int[][] matrix) {
        int m = matrix.length, n = matrix[0].length;
        boolean isPrimary = true;
        boolean isSecondary = true;

        Map<Integer, Integer> primaryMap = new HashMap<>();
        Map<Integer, Integer> secondaryMap = new HashMap<>();

        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {

```

```

        int keyP = i - j; // key for primary diagonals
        int keyS = i + j; // key for secondary diagonals

        // Primary
        if (!primaryMap.containsKey(keyP)) {
            primaryMap.put(keyP, matrix[i][j]);
        } else if (primaryMap.get(keyP) != matrix[i][j]) {
            isPrimary = false;
        }

        // Secondary
        if (!secondaryMap.containsKey(keyS)) {
            secondaryMap.put(keyS, matrix[i][j]);
        } else if (secondaryMap.get(keyS) != matrix[i][j]) {
            isSecondary = false;
        }
    }

    if (isPrimary && i

```

Complexity:

- Time: $O(m * n)$
 - Space: $O(m * n)$
-

5. Justification / Proof of Optimality

- Brute Force â†’ simple & optimal for small 20x20 matrices.
 - HashMap â†’ scalable if matrix grows larger (e.g., streaming input).
-

6. Variants / Follow-Ups

- Strictly Primary only Toeplitz check (original problem).
- Strictly Secondary only Toeplitz check.
- Check if matrix is Hankel matrix (same as Secondary Toeplitz).
- Allow up to k mismatches.
- Generate Toeplitz matrix programmatically.

Q34: Diagonal Traversal Function (Top-Right & Top-Left)

1. Understand the Problem

- **Paraphrase:** We start from either top-right corner or top-left corner, and move diagonal by diagonal until the end.
-

2. Constraints

- $1 \leq N \leq 500$
 - $-10^4 \leq \text{mat}[i][j] \leq 10^4$
-

3. Examples & Edge Cases

Example 1 (Top-Right â†’ Bottom-Left): Input:

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

Output:

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

Example 2 (Top-Left â†’ Bottom-Right): Input:

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

Output:

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

4. Approaches

Approach 1: Brute Force Using Extra Storage

- **Idea:**

- Store each diagonal in a list and print later.

Java Code:

```
List<Integer> result = new ArrayList<>();
// Traverse diagonals, store elements in result
// Print result at end
```

Complexity:

- Time: $O(N^2)$ “ every element visited once.
- Space: $O(N^2)$ “ storing traversal in a list.

Approach 2: Optimal In-Place Traversal

- **Idea:**

- Idea: Print elements while traversing diagonals without extra memory.
- Steps:
 - For Top-Right †’ Bottom-Left:
 - Start from top row, last column †’ traverse diagonals.
 - Then start from first column, row 1 †’ traverse remaining diagonals.
- For Top-Left †’ Bottom-Right:
 - Start from top row, first column †’ traverse diagonals.
 - Then start from last column, row 1 †’ traverse remaining diagonals.

Java Code:

```
public static void diagonalTraversalInPlace(int[][] mat, int n, String direction)
{
    if(direction.equalsIgnoreCase("topRight")) {
        for(int col=n-1; col>=0; col--) {
            int i=0, j=col;
            while(i<n && j<n) { System.out.print(mat[i][j]+" "); i++; j++; }
        }
        for(int row=1; row<n; row++) {
            int i=row, j=0;
            while(i<n && j<n) { System.out.print(mat[i][j]+" "); i++; j++; }
        }
    } else if(direction.equalsIgnoreCase("topLeft")) {
        for(int col=0; col<n; col++) {
            int i=0, j=col;
            while(i<n && j>=0) { System.out.print(mat[i][j]+" "); i++; j--; }
        }
        for(int row=1; row<n; row++) {
            int i=row, j=n-1;
            while(i<n && j>=0) { System.out.print(mat[i][j]+" "); i++; j--; }
        }
    }
}
```

Complexity:

- Time: $O(N^2)$
 - Space: $O(1)$ (no extra list)
-

5. Justification / Proof of Optimality

- In-place solution is optimal: no extra memory, linear traversal of all elements.
 - Brute-force using extra list is unnecessary for large N.
 - Complexity is optimal for $N \times N$ matrix: $O(N^2)$ time, $O(1)$ space.
-

6. Variants / Follow-Ups

- Non-square matrix ($M \times N$) → can adapt traversal loops.
- Anti-diagonal traversal → starting from bottom-right instead of top corners.
- Diagonal sum instead of traversal → compute sums during traversal.

Q44: Special Matrix

1. Understand the Problem

- **Read & Identify:** Determine whether a given square matrix is a special matrix. Definition: Diagonal elements ($\text{matrix}[i][i]$) are non-zero. All non-diagonal elements ($\text{matrix}[i][j]$ where $i \neq j$) are zero.
 - **Goal:** Return true if the matrix satisfies the above conditions, otherwise false.
 - **Paraphrase:** Only diagonal elements are non-zero; everything else must be zero.
-

2. Constraints

- $1 \leq T \leq 10$
 - $1 \leq N \leq 200$
 - $0 \leq A[i][j] \leq 10^6$
-

3. Examples & Edge Cases

Example 1 (Normal Case): Input:

```
1
3
1 0 2
```

```
0 2 0  
3 0 1
```

Output:

```
true
```

Example 2 (Normal Case): Input:

```
1  
3  
1 0 1  
1 2 0  
2 0 3
```

Output:

```
false
```

4. Approaches

Approach 1: Brute Force

- **Idea:**
 - Traverse the entire matrix.
 - For each element $\text{matrix}[i][j]$:
 - If $i == j$ → check non-zero
 - Else → check zero

Java Code:

```
public boolean isSpecialMatrix(int[][] matrix) {  
    int n = matrix.length;  
    for (int i = 0; i < n; i++) {  
        for (int j = 0; j < n; j++) {  
            if (i == j) {  
                if (matrix[i][j] == 0) return false;  
            } else {  
                if (matrix[i][j] != 0) return false;  
            }  
        }  
    }  
}
```

```
        return true;
    }
```

Complexity:

- Time: $O(N^2)$ to visit all elements.
- Space: $O(1)$ to in-place.

Approach 2: Optimized (Early Exit)

- **Idea:**
 - Traverse row by row.
 - As soon as a non-diagonal element is non-zero or a diagonal element is zero → return false.
 - Otherwise, after traversal → return true.

Java Code:

```
public boolean isSpecialMatrixOptimized(int[][] matrix) {
    int n = matrix.length;
    for (int i = 0; i < n; i++) {
        if (matrix[i][i] == 0) return false; // diagonal must be non-zero
        for (int j = 0; j < n; j++) {
            if (i != j && matrix[i][j] != 0) return false; // non-diagonal must be
zero
        }
    }
    return true;
}
```

Complexity:

- Time: $O(N^2)$ in worst-case, but can exit early.
- Space: $O(1)$ to in-place.

5. Justification / Proof of Optimality

- Both approaches correctly check all diagonal and non-diagonal conditions.
- Optimized approach can exit early, saving some comparisons when matrix is not special.

6. Variants / Follow-Ups

- Check if a matrix is diagonal (diagonal elements can be zero).
- Check for identity matrix (diagonal elements must be 1, others 0).
- Special matrices with non-square matrices → check main diagonal only.