**QPREP3-Matrix traversal**

**Module Introduction**

#### Write a program to find the value at a position in the matrix given the current position, the direction to move and number steps to move.

#### Objective

A matrix and a valid current position (row number and column number) are provided as input. Given a direction (RIGHT is 1, DOWN in 2, LEFT is 3 and UP is 4) and number of steps to move, output the values along the path taken to the new position.

If any position during traversal is out of bounds, including the initial position, output a single -1 in the list and not the values along the path. The matrix is guaranteed to have non-negative numbers. The direction will always be valid and the number of steps will always be > 0.

#### Input Format & Example

**Example 1**

Input:

3 3 --> Matrix size

1 2 3 --> Input matrix of 3 rows

4 5 6

7 8 9

1 1 --> Current Position

1 1 --> Direction, # of steps; In this case move RIGHT by 1 step

Output:

6

**Example 2**

Input:

3 4

1 2 3 4

5 6 7 8

9 10 11 12

0 2

2 2

Output:

7 11

***SOLUTION STEPS FROM NEXT PAGE:***

**Write down at least 3 examples in the following format. Kindly stick to the format.**

**Suggestion:**

EXAMPLE#1

INPUT:

3 2

1 2

4 5

6 7

2 0

4 2

OUTPUT:

4 1

EXAMPLE#2

INPUT:

4 4

4 5 6 7

1 2 3 4

9 8 7 6

5 4 3 2

2 3

3 3

OUTPUT:

7 8 9

EXAMPLE#3

INPUT:

2 2

4 5

1 2

0 0

1 1

OUTPUT:

5

**Detail your problem understanding here**

**Suggestion:**

There are multiple ranges provided as input. Each range will have a starting and ending number.

The solution should go through each of the ranges. For any overlapping ranges, they should be merged into a single range. The merged range will have the starting number as the minimum of all the starting numbers of the overlapping ranges and the ending number as the maximum of all the ending numbers.

We may need to create multiple consolidated ranges created across the ranges provided.

There may be multiple ranges that converge to a single range. If there is no overlap for some of the ranges, keep them as is.

Output the consolidated ranges. The matrix is provided as input, with m x n rows.

The current position is provided as row i and column j.

The direction to move is specified as UP, DOWN, LEFT or RIGHT and the number of steps to move in that direction is also specified.

We have to traverse the numbers in the matrix from the current position in a specific direction given number of times. Return the numbers visited along the way. If the movement takes us beyond the matrix’s boundary, print -1.

**Does this problem follow a known algorithmic pattern or standard application of a data structure? If there are multiple approaches, which one would you choose and why? Write down your chosen approach in 2-3 sentences like you would explain to a 10 year old.**

The problem seems a straight forward matrix traversal. Moving horizontally involves changing ‘column’ and moving vertically involves moving ‘row’. Need to ensure row or column is not out of bounds.

**Write the pseudocode here in plain English**

moveRight(x, y, steps) - Move right by incrementing y until y < columnEnd

moveLeft(x, y, steps) - Move left by decrementing y until y >= 0

Similarly write moveUp() & moveDown()

Read the matrix, the current position, direction to move and steps to move

Depending on the direction to move, invoke the corresponding move function above

Pass the current position (x and y), matrix, matrix size (m and n) and steps to move

**Can you specify a few boundary or edge cases here?**

**Edge cases**

Input:

3 4

1 2 3 4

5 6 7 8

9 10 11 12

0 2

2 5

Output:

-1

**Write the functions you would create here**

List moveLeft (int m, int n, int[ ][ ] matrix, int x, int y, int steps)

List moveRight (int m, int n, int[ ][ ] matrix, int x, int y, int steps)

List moveDown (int m, int n, int[ ][ ] matrix, int x, int y, int steps)

List moveUp (int m, int n, int[ ][ ] matrix, int x, int y, int steps)

List matrixMove(int m, int n, int[ ][ ] matrix, int x, int y, int direction, int steps)

#### Summary

Starting with a brief explanation of the problem statement followed by pseudocode and then implementing the solution helps you approach the problem in a systematic way. This methodology helps with easy as well as hard problems.

**Time Complexity: O(1)**

The movement to the new position doesn’t need additional steps.

**Space Complexity: O(1)**

No new storage is being used to implement the solution.

#### Concepts

Concepts covered in this Module

* Array
* Matrix

Similar problems

* <https://leetcode.com/problems/search-a-2d-matrix/>
* <https://leetcode.com/problems/minimum-time-visiting-all-points/>

#### Good habits

Think about these for your solution:

* Comments - have you used comments in a way that others can understand this code?
* Test Cases - Are most of the scenarios/corner cases/boundary conditions handled in the solution?
* Naming Convention - Are the variables and functions named sensibly and with uniform convention?
* Modular Functions - Has the solution been addressed using concise functions? Will these functions work without any changes if they are to be used in another problem?
* Optimization - Analyze the Time Complexity and Space Complexity for your solution. Has the solution been optimized or did it use the brute force method? Is further optimization desirable/possible?
* Data Structures - Has the optimal/appropriate data structure been used?

SOLUTION:

APPROACH 1: solved. Can be more modular ofc.

import java.util.\*;

class MatrixTraversal {

static boolean isSafe(int x, int y, int m, int n)

{

return (x >= 0 && x < m && y >= 0 && y < n);

}

// complete the below function implementation

public List<Integer> valueAtNewPosition(int[][] matrix, int currX, int currY, int dir, int steps) {

List<Integer> lst = new ArrayList<Integer>();

int flag=0;

// (RIGHT is 1, DOWN in 2, LEFT is 3 and UP is 4)

if(dir==1)

{

for(int i = 1;i<=steps;i++)

{

if (! isSafe(currX, currY+i, matrix.length, matrix[0].length))

{

flag=1;

break;

}

else

{

lst.add(matrix[currX][currY+i]);

}

}

}

else if(dir==2)

{

for(int i = 1;i<=steps;i++)

{

if (! isSafe(currX+i, currY, matrix.length, matrix[0].length))

{

flag=1;

break;

}

else

{

lst.add(matrix[currX+i][currY]);

}

}

}

else if(dir==3)

{

for(int i = 1;i<=steps;i++)

{

if (! isSafe(currX, currY-i, matrix.length, matrix[0].length))

{

flag =1;

break;

}

else

{

lst.add(matrix[currX][currY-i]);

}

}

}

else if(dir==4)

{

for(int i = 1;i<=steps;i++)

{

if (! isSafe(currX-i, currY, matrix.length, matrix[0].length))

{

flag=1;

break;

}

else

{

lst.add(matrix[currX-i][currY]);

}

}

}

if(flag==1)

{

List<Integer> result = new ArrayList<Integer>();

result.add(-1);

return result;

}

return lst;

}

public static void main(String args[]) {

Scanner scanner = new Scanner(System.in);

int n = scanner.nextInt();

int m = scanner.nextInt();

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

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

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

matrix[i][j] = scanner.nextInt();

}

}

int currPosX = scanner.nextInt();

int currPosY = scanner.nextInt();

int dirToMove = scanner.nextInt();

int stepsToMove = scanner.nextInt();

scanner.close();

List<Integer> result = new MatrixTraversal().valueAtNewPosition(matrix, currPosX, currPosY, dirToMove, stepsToMove);

for (int i = 0; i < result.size(); ++i) {

System.out.printf("%d ", result.get(i));

}

}

}

**Complexity Analysis:**

* **Time Complexity:**
* **Space Complexity:**