Sample Input	Sample Output

Multidimensional Arrays [CO1]

[Each method carries 5 marks]

Instructions for students:

- Complete the following methods on 2D Arrays
- You may use any language to complete the tasks.
- All your methods must be written in one single .java or .py or .pynb file. DO NOT CREATE separate files for each task.
- If you are using JAVA, you must include the main method as well which should test your other methods and print the outputs according to the tasks.
- If you are using PYTHON, then follow the coding templates shared in this folder.

NOTE:

- **YOU CANNOT USE ANY BUILT-IN FUNCTION EXCEPT** len **IN PYTHON. [negative indexing, append is prohibited]**
- **You can use the attribute 'shape' of numpy arrays**
- **YOU HAVE TO MENTION SIZE OF ARRAY WHILE INITIALIZATION**
- **YOUR CODE SHOULD WORK FOR ANY VALID INPUTS. [Make changes to the Sample Inputs and check whether your program works correctly]**

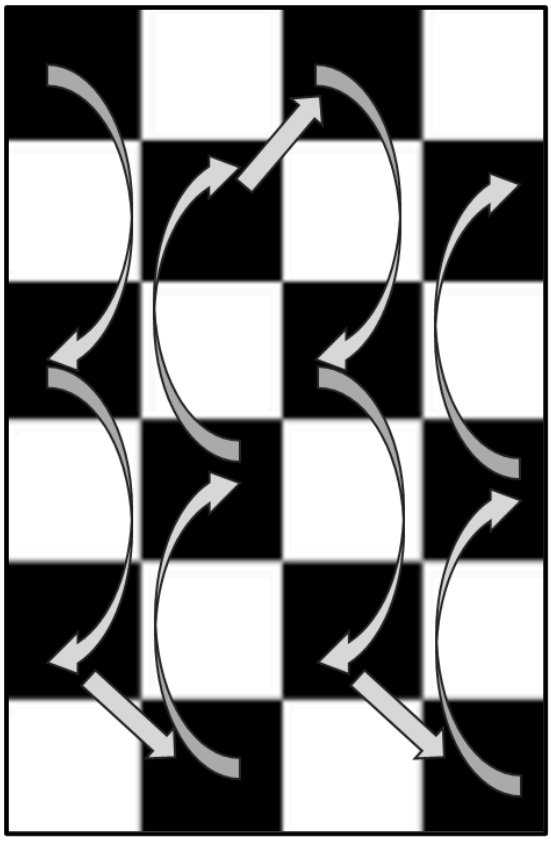
Sample Input	Sample Output

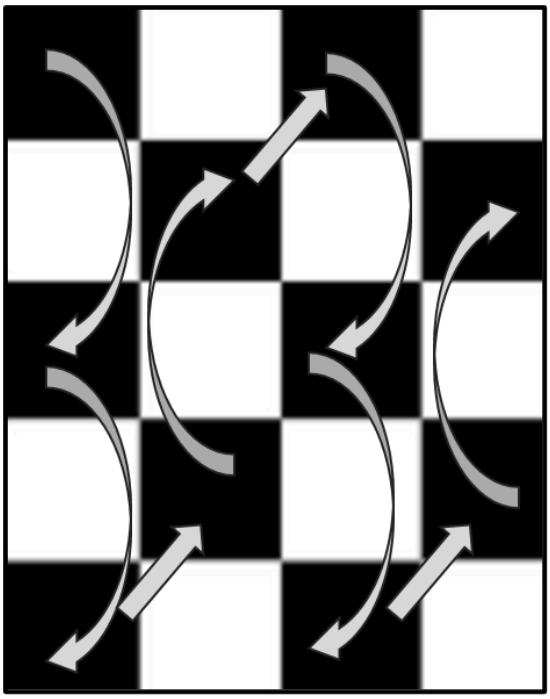
2D Array tasks:

1. Zigzag Walk:

As a child, you often played this game while walking on a tiled floor. You walked avoiding a certain color, for example white tiles (it almost felt like if you stepped on a white tile, you would die!). Now you are in a room of m x n dimension. The room has m*n black and white tiles. You step on the black tiles only. Your movement is like this:

OR





Now suppose you are given a 2D array which resembles the tiled floor. Each tile has a number. Can you write a method that will print your walking sequence on the floor?
Constraint: The first tile of the room is always black.

Sample Input	Sample Output

Hint: Look out for the number of rows in the matrix [Notice the transition from column 0 to column 1 in the above figures]

---------- 3 | 8| | 4 | 6 | 1 | ---------- | 7 | 2 | 1 | 9 | 3 | ---------- | 9 | 0 | 7 | 5 | 8 | ---------- | 2 | 1 | 3 | 4 | 0 | ---------- | 1 | 4 | 2 | 8 | 6 | ----------	3 9 1 1 2 4 7 2 4 9 1 8 6
---------- | 3 | 8 | 4 | 6 | 1 | ---------- | 7 | 2 | 1 | 9 | 3 | ---------- | 9 | 0 | 7 | 5 | 8 | ---------- | 2 | 1 | 3 | 4 | 0 | ----------	3 9 1 2 4 7 4 9 1 8

Sample Input	Sample Output

2. Row Rotation Policy of BRACU Classroom:

You are no longer permitted to choose your own seat on the new campus of BRACU. You must abide by the new regulations, which state that if you sit in the first row for the first week, you must shift to the second row for Week2, the third row for the week3, and so on. In your classroom, there are a total of 6 rows and 5 columns. Your friend "AA" wants to know from you that on the upcoming exam week in which row he will be in. Your task is to implement a function row_rotation(exam_week, seat_status) that takes the exam_week and a 2D array of current seat status as input and returns the row number in which your friend "AA" will be seated and print the seat status for that week.

Input: exam_week = 3
current_seat_status=

A	B	C	D	E
F	G	H	I	J
K	L	M	N	O
P	Q	R	S	T
U	V	W	X	Y
Z	AA	BB	CC	DD

Output:

U	V	W	X	Y
Z	AA	BB	CC	DD
A	B	C	D	E
F	G	H	I	J
K	L	M	N	O
P	Q	R	S	T

Your friend AA will be on row 2.

Explanation: exam is 3 weeks away from the current week
Current seat status of above input is for this week.

Seat Status of Next week:

Z	AA	BB	CC	DD
A	B	C	D	E
F	G	H	I	J

Seat Status of Exam week:

U	V	W	X	Y
Z	AA	BB	CC	DD
A	B	C	D	E

Sample Input	Sample Output

K	L	M	N	O
P	Q	R	S	T
U	V	W	X	Y

F	G	H	I	J
K	L	M	N	O
P	Q	R	S	T

3. Matrix Manipulation:

You are participating in a coding competition focused on matrix manipulation, and your task is to develop a Python function that reverses the rows and columns of a square matrix. Your function should take a square matrix of size N x N as input and return a new matrix where the rows and columns are reversed.

Write a method named reverse_Matrix() that accepts a 2D numpy array representing the matrix as input and returns a new matrix with reversed rows and columns. The function should adhere to the following specifications:

- The input matrix will be a square matrix, meaning it will have the same number of rows and columns.
- The function should not modify the original matrix; instead, it should create and return a new matrix.
- The new matrix should have the same size as the input matrix.
- The rows and columns of the new matrix should be in reverse order compared to the input matrix.

Sample Input	**Output**
| 8 | 10 | 2 | ---------- | 3 | 14 | 14 | ---------- | 0 | 4 | 7 | ----------	| 7 | 4 | 0 | ---------- | 14 | 14 | 3 | ---------- | 2 | 10 | 8 | ----------
| 14 | 8 | 0 | 4 | ---------- | 9 | 8 | 13 | 13 |	| 6 | 13 | 10 | 2 | ---------- | 4 | 1 | 3 | 9 |

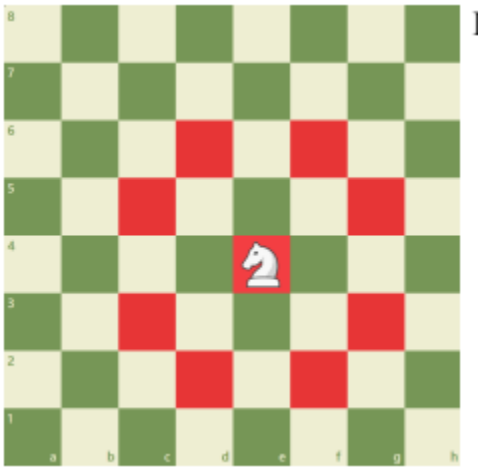
Sample Input	Sample Output
---------- | 9 | 3 | 1 | 4 | ---------- | 2 | 10 | 13 | 6 | ----------	---------- | 13 | 13 | 8 | 9 | ---------- | 4 | 0 | 8 | 14 | ----------

4. **Chess Piece:**

The chess board is 8x8 size. We will only deal with the knight piece here.

The knight piece moves like this-
From its position, it can move in 8 ways:

a. 2 cells in upward direction and 1 cell in right direction.
b. 2 cells in upward direction and 1 cell in left direction.
c. 2 cells in downward direction and 1 cell in right direction.
d. 2 cells in downward direction and 1 cell in left direction.
e. 2 cells in the left direction and 1 cell in upward direction.
f. 2 cells in the left direction and 1 cell in downward direction.
g. 2 cells in the right direction and 1 cell in upward direction.
h. 2 cells in the right direction and 1 cell in downward direction.



Sample Input	Sample Output

Given the position of a knight in a tuple, your task is to write a method that calculates all the possible moves of the knight and return a 8x8 chessboard where empty cells are 0 and the probable knight moves are denoted as 3. The given knight cell has 66 in it.
Be very careful with CORNER CASES.

Sample Input:	Sample Output:
knight = (3,4)	| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | **3** | 0 | **3** | 0 | 0 | --- | 0 | 0 | **3** | 0 | 0 | 0 | **3** | 0 | --- | 0 | 0 | 0 | 0 | **66** | 0 | 0 | 0 | --- | 0 | 0 | **3** | 0 | 0 | 0 | **3** | 0 | --- | 0 | 0 | 0 | **3** | 0 | **3** | 0 | 0 | --- | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ---

Sample Input	Sample Output

5. Matrix Compression:

Complete the function **compress_matrix** that takes a 2D array as a parameter and return a new compressed 2D array. In the given array the number of row and column will always be even. **Compressing a matrix means grouping elements in 2x2 blocks and sums the elements within each block. Check the sample input output for further clarification.**

Hint: Generally the block consists of the (i,j), (i+1,j), (i,j+1) and (i+1, j+1) elements for 2x2 blocks.

You cannot use any built-in function except len() and range(). You can use the np variable to create an array.

Python Notation	Java Notation

import numpy as np
def compress_matrix (mat):
    # To Do
	public int[ ][ ] compress_matrix (int[] [] mat) {
	    // To Do
	}

Sample Input array	All Box (No need to create these arrays)	Returned Array	Explanation
[[1, 2, 3, 4], [5, 6, 7, 8], [1, 3, 5, 2], [-2, 0, 6,-3]]	[[1, 2], [[3, 4], [5, 6]] [[7, 8]] [[1, 3], [[5, 2], [-2, 0]] [[6, -3]]	[[14, 22], [2, 10]]	[[1+2+5+6, 3+4+7+8], [1+3+-2+0, 5+2+6+-3]]

Sample Input	Sample Output

6. Game Arena:

Suppose you and your friends are in the world of '*Alice in Borderland*' where you decided to take part in a game and entered the *game arena*. As a team, you need to gain **at least 10 points** in order to keep surviving in the borderland. Otherwise, you will be out of the game and your team will be banished for good. Now, the arena has a 2D array like structure where players of a team are given certain positions with values that are **multiples of 50**. By staying in these positions, every player can gain points from the cells above, below, left and right (not diagonally) only if those cells **contain 2** [The cells containing 1s and 0s are to be avoided]. For each player, add from these cells containing 2s to your total points for the team to keep on surviving in borderland. **Be careful about corner cases. Your task is to write a method which tells us whether your team is out or has survived the game.**

Sample Input 1	**Sample Output 2**
---------- **| 0 | 2 | 2 | 0 |** **----------** **| 50 | 1 | 2 | 0 |** **----------** **| 2 | 2 | 2 | 0 |** **----------** **| 1 | 100 | 2 | 0 |** **----------**	**Points Gained: 6. Your team is out.**
Explanation: Player with value 50 has 2 in the cell below him (1 cell). Player with value 100 has 2 in the cell above and in the right cell (2 cells). So in total, they got (1+2)*2 = 6 points which was not enough to survive the game.	

Sample Input 1	**Sample Output 2**
---------- **| 0 | 2 | 2 | 0 | 2 |** **----------** **| 1 | 50 | 2 | 1 | 100 |** **----------** **| 2 | 2 | 2 | 0 | 2 |** **----------**	**Points Gained: 14. Your team has survived the game.**

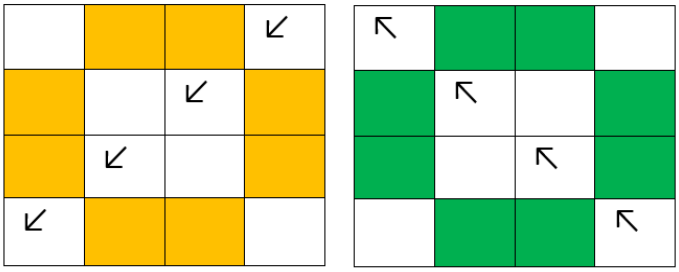
Sample Input	Sample Output

| 0 | 200 | 2 | 0 | 0 | **----------**	

Explanation:
Player with value 50 has 2 in the cell above, cell below and the right cell (3 cells). Player with value 100 has 2 in the cell above and the cell below (2 cells). Player with value 200 has 2 in the above cell and right cell (2 cells). So in total, they gained (3+2+2)*2 = 14 points and survived the game.

Note: For the cell with value 2 that is common between 2 players in position (2,1), both gained 2 points each, so it's not like if one player already added those 2 points, another player cannot.

Bonus Task: Primary vs Secondary Diagonal



Write one function named **check_Diagonal()** that takes two NxN 2D arrays as parameters to check the **following point** and print “YES” if the point is true; otherwise, “NO”. [Value of N >= 3]. The dimensions of both arrays will always be equal.

The point is:

The first array's secondary diagonal elements from the top right corner to the bottom left corner equals the second array's primary diagonal elements from the bottom right corner to the top left corner.

Example:

Sample Input	Sample Output

First Array:

0	9	9	1
7	0	2	6
7	3	0	6
4	8	8	0

Second Array:

4	8	8	0
6	3	0	7
7	0	2	7
0	9	9	1

Explanation:

The first array's **secondary diagonal** values are 1, 2, 3, 4 at (0, 3), (1, 2), (2, 1), and (3, 0) indexes.
The second array's **primary diagonal** values are 1, 2, 3, 4 at (3, 3), (2, 2), (1, 1), and (0, 0) indexes.

Sample Input:	**Sample Output:**
array1 = np.array([[0, 4, 1], [7, 2, 5], [3, 6, 0]]) array2 = np.array([[3, 6, 0], [5, 2, 7], [0, 4, 1]]) check_Diagonal (array1, array2)	**YES**
array1 = np.array([[0, 9, 9, 1], [9, 0, 2, 9], [9, 3, 0, 9], [4, 9, 9, 0]]) array2 = np.array([[4, 9, 9, 0], [9, 0, 3, 9], [9, 0, 2, 9], [0, 9, 5, 1]]) check_Diagonal (array1, array2)	**NO**

Sample Input	Sample Output