The Galactic Chessmaster's Challenge

In the year 3045, the Intergalactic Chess Federation (ICF) faces a unique challenge. The reigning champion, Grandmaster Zara Nova, has proposed a revolutionary twist to the ancient game: Quantum Chess, played on an n x n board where n can vary from match to match.
In Quantum Chess, each of the n x n square's "quantum state" is represented by a number. Before each match, the board must be prepared following two crucial steps:

1. **Row Stabilization**: Each row of the board must be sorted in ascending or descending order of quantum states, ensuring stable quantum entanglement within rows.
2. **Inversion Calculation**: In this challenge, an inversion refers to the instance where, in the board, a square with a higher (strictly greater) quantum state precedes a square with a lower quantum state. The total inversion value is crucial for determining the match's difficulty level.

Meanwhile, the ICF is also planning the upcoming Galactic Chess Olympics. They face a logistical challenge of pairing players for the opening rounds. The players are spread across the vast Olympic Space Station which is represented as a 2D plane. Each player's coordinates are available with the ICF. To start the game, the ICF needs to find the **pair of players with the closest distance** (least Euclidean distance).

With this setting, the ICF's tech team, led by the brilliant Dr. Axel Quark, must develop a system to do the following:

1. Sort the quantum states of each row on the n x n Quantum Chess board.
2. Calculate the total inversion across the entire board.
3. Find the closest pair of players on the 2D map of the space station to start the match

Requirements

You can create whatever classes you want.
You need to create the following methods:

- `sortRows(Comparator comparator)`: Sorts the matrix row-wise using a custom functor Comparator (ascending or descending - this will be mentioned in the test case).
  - *Arguments:* A `Comparator` object.
- `countInversions()`: An inversion is the phenomenon where a strictly greater valued element is present at a smaller index in the matrix compared to a greater valued element

  This function countInversions() counts and returns the number of inversions in the matrix. The inversions are counted by flattening the 2D matrix into a 1D array, i.e. the inversions will be counted across the whole matrix.

  - For example, let the matrix be

    1 3

    3 2

    So by flattening it we get the 1d array 1 3 3 2.

    For index 0, #inversions = 0 (Since element at index 0 is not strictly greater than any of the elements at an index greater than 0).

    For index 1, #inversions = 1 (Since element at index 1 is strictly greater than the element at index 3).

    For index 2, #inversions = 1 (Since element at index 2 is strictly greater than the element at index 3).

    For index 3, #inversions = 0 (Since element at index 3 is not strictly greater than any of the elements at an index greater than 3).

    Hence total inversions = 2

    The function countInversions() should return 2 for this matrix

- `display()`: Prints the matrix.
- `closestPair()`: Finds and returns the coordinates of the pair of points with the least Euclidean distance. If the distance between two different pairs are the same, then return the pair with the lesser value of x coordinate. If both distance and x coordinate are same, then return the pair with lesser value of y coordinate.

Input Format

The following commands will be given as input:

1. **Create 2D Matrix:**
    - Command: `CREATE_2D size`
    - Followed by `size` lines each having `size` integers each denoting the quantum states.
2. **Sort a Matrix:**
    - Command: `SORT_2D ascending` ○ Command: `SORT_2D descending`
3. **Count Inversions:**
    - Command: `INVERSION_2D`
4. **Display Matrix:**
    - Command: `DISPLAY_2D`
5. **Closest Pair:**
    - Command: `CLOSEST_2D num_points`
    - Followed by `num_points` lines each having two space separated integers denoting the coordinates of the players

The input will end with the `END` command.

Output Format

1. The `DISPLAY_2D` command should output the matrix in a row-wise manner. Each row will be printed on a new line with each element of a row being separated by a space " ".
2. The `INVERSION_2D` command will output a single integer on a newline indicating the number of inversions in the matrix.
3. The `CLOSEST_2D` command will output four integers on a single line separated by a space " ". The first two integers denote the coordinates of the first point and the next two integers denote the coordinates of the second point. The points are ordered in the same order as they appear in the input. For example, if the two points are P1 (x1, y1) and P2 (x2, y2) where the point P1 appears before the point P2 in the input, then the output of CLOSEST_2D will be like:

x1 y1 x2 y2

Example Input

```
CREATE_2D 4
1 2 2 1
2 1 1 2
3 4 3 4
4 3 3 4
DISPLAY_2D
INVERSION_2D
SORT_2D ascending
DISPLAY_2D
CLOSEST_2D 4
8 7
-3 9
-5 -9
-10 7
END
```

Expected Output

```
1 2 2 1
2 1 1 2
3 4 3 4
4 3 3 4
15
1 1 2 2
1 1 2 2
3 3 4 4
3 3 4 4
-3 9 -10 7
```

Explanation

The matrix is given input by the CREATE_2D command. The input matrix is a 4 x 4 matrix of the form:

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

The DISPLAY_2D command displays this output (which are the first 4 lines in the output).

The next command is INVERSION_2D. So we flatten the matrix as a 1D array like: 1 2 2 1 2 1 1 2 3 4 3 4 4 3 3 4. Here we calculate the inversions for each index.

Element :	1	2	2	1	2	1	1	2	3	4	3	4	4	3	3	4

Inversions: 0 3 3 0 2 0 0 0 0 3 0 2 2 0 0 0
Total inversions = 15
Hence the next line of output is 15.

The next command is SORT_2D which sorts the matrix row wise. So the matrix now becomes:

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

The next command is DISPLAY_2D which outputs this matrix (which are the next 4 lines in the output).

The next command is CLOSEST_2D with an integer 4. Hence the next 4 lines in the input are 4 pairs of integers denoting the coordinates of the 4 points in a 2d plane.
We find the pairwise Euclidean distance among the points. And we get that the points with coordinates (-3, 9) and (-10, 7) are having the least distance. Hence we output these coordinates in the same order as they appear in the input (which is the next line in the output).

Constraints

1 <= dimension_of_matrix <= 10^3

-10^18 <= quantam_values <= 10^18 - 1

-10^9 <= coordinate_values_of_points <= 10^9

We will be working with only one matrix per test case (i.e. CREATE_2D command will appear only once).

There can be any number of CLOSEST_2D commands in a test case (followed by their respective number of points). But the total number of pairs including all the CLOSEST_2D commands in a test case will be <= 10^5

The DISPLAY_2D command will be called at max 3 times within a test case.

The total number of commands (including all of CREATE_2D, CLOSEST_2D, DISPLAY_2D, INVERSION_2D, SORT_2D) per test case will be maximum 50.

Note

You are expected to use the **divide and conquer** approach while designing your algorithms for all the problems .

The tie breakers in case of closest pair should be solved as mentioned below

1. By ascending squared distance between points
2. If distances are equal, by ascending x-coordinate of the first point
3. If first x-coordinates match, by ascending y-coordinate of the first point
4. If first points are identical, by ascending x-coordinate of the second point
5. If second x-coordinates match, by ascending y-coordinate of the second point
6. If all coordinates are equal, maintain the original input order using point indices