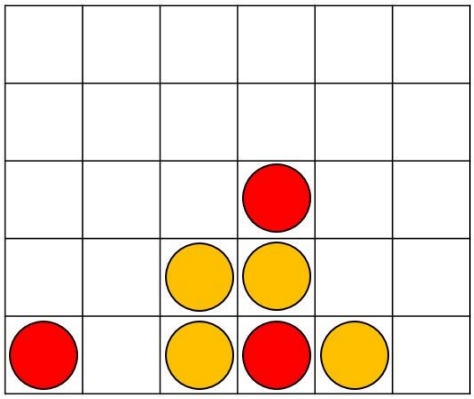
Four Up (17 marks, Suggested Time: 80 min)

After spending months completing a project, team leader Ryan decides to assign Jed and Jared to a new project. MINIONS.pte.ltd has just taken on a new project called Four Up.

Upon receiving the news, Jared has had enough of this and decided to quit the company leaving only Jed to be in charge. Jed then approached you to help him, which you had no choice but to agree. However after a few days, you received news that you are promoted to Team Leader with a pay raise of ($50). This is because Jed also decided to quit leaving you alone with this project.

Four Up is a two player game played using a vertically suspended grid. Players take turns dropping their tokens into a column of the grid. The tokens will fall straight down, occupying the bottom most available row of the column. For example, in the figure below, a token dropped into the third column from the left will end up on the third row from the bottom.



(4, 0)	(4, 1)	(4, 2)	(4, 3)	(4, 4)	(4, 5)
(3, 0)	(3, 1)	(3, 2)	(3, 3)	(3, 4)	(3, 5)
(2, 0)	(2, 1)	(2, 2)	(2, 3)	(2, 4)	(2, 5)
(1, 0)	(1, 1)	(1, 2)	(1, 3)	(1, 4)	(1, 5)
(0, 0)	(0, 1)	(0, 2)	(0, 3)	(0, 4)	(0, 5)

A player wins the game if they are the first to form a continuous horizontal, vertical or diagonal line of at least four of their own tokens.

You are to represent the game state using a list of lists. Each element of the list represents a row of the grid, which is a list of positions on the board. An empty spot on the grid can be represented by None, and a player's token can be a string or integer. For example, one player might use the string "P1" and the other might use the integer 42 as their tokens. You may assume the two players will not use the same token.

[4a] The function make_grid takes as inputs the number of rows and columns of a playing grid, and returns an empty grid of the given size, represented as a list of lists as stated above. Provide an implementation for make_grid.
[2m]

From now on, the following functions are provided to you which you can call.

- **rows(grid)** takes in a grid and returns the number of rows.
- **cols(grid)** takes in a grid and returns the number of columns.
- **get_value(grid, row, col)** returns the value at the given row and column of the grid. If the given coordinates are outside the grid, None is returned.
- **set_value(grid, row, col, val)** updates the element at the given row and column of the grid to val and returns the grid.

[4b] Implement the function drop, which takes in the game grid, a column number, and a player's token. It updates the grid to the new state where the player makes a move by dropping their token in the given column. If the stated column is already filled to top, a ColumnFullError(col) is raised.

Assume the ColumnFullError class has already been defined and that the row and column numbering starts from 0.
[2m]

[4c] It is quite straightforward to check if a player has formed a continuous horizontal row of four of the same token in a grid by writing a function check_rows as follows:

```
# The following function should not be changed
def check_rows(grid):
    for row in grid:
        token = check_row(row) # check individual row
        if token:
            return token
    return False
```

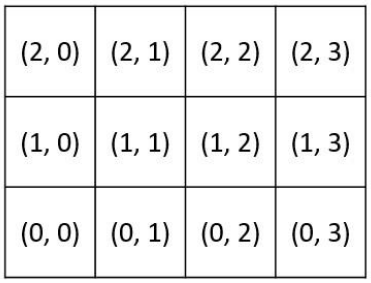
The function check_row takes a single row of the grid as input, and returns the token that occupies at least four continuous adjacent positions in the row. If no such token exists, False is returned.

Provide an implementation of the function check_row. You may assume that at most only one

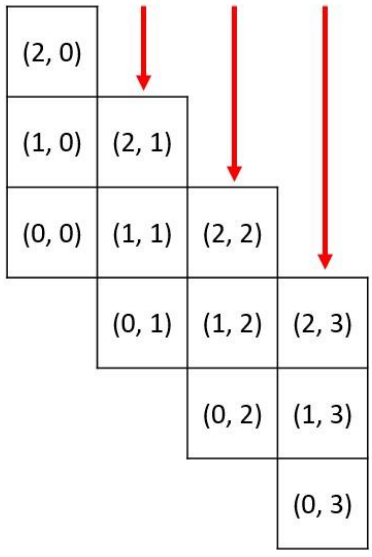
token will satisfy the condition.
[2m]

[4d] It is also straightforward to check for any continuous vertical connections. We can simply transpose or rotate the grid like a matrix and since the columns now become the rows, the check_rows function can be reused. However, it is not so simple to check the diagonals.

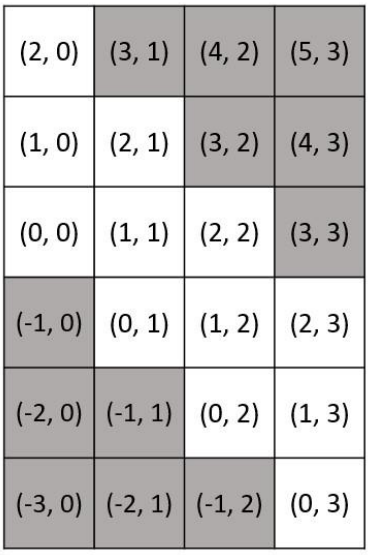
After a month into the project, Ryan employs a new minion (Marcus). Marcus thinks there is a way to reuse the check_rows function if we can somehow skew the grid by 45 degrees, as shown below:



Example of a 3 x 4 grid



Skewing the grid by 45°



Final skewed grid

A grid skewed by 45 degrees will result in the diagonals becoming rows. The shaded cells indicates cells that are not part of the original grid. Implement the function diagonal, which takes in a grid and outputs the grid that is skewed by 45 degrees. **[5m]**

For Example:

```
>>> diagonal(grid)
[ [(2, 0), (3, 1), (4, 2), (5, 3)],
  [(1, 0), (2, 1), (3, 2), (4, 3)],
  [(0, 0), (1, 1), (2, 2), (3, 3)],
  [(-1, 0), (0, 1), (1, 2), (2, 3)],
  [(-2, 0), (-1, 1), (0, 2), (1, 3)],
  [(-3, 0), (-2, 1), (-1, 2), (0, 3) ] ]
```

[4e] In order to complete this task, Marcus needs another function deep_map. deep_map(fn, l) takes as inputs a function and a list of nested lists. It returns a new list which has all its elements and nested elements modified via the function fn.

For example:

```
>>> l = [1, 2, [3, 4], [5, 6, [7]]]
>>> deep_map(lambda x: x*2, l)
>>> l
[2, 4, [6, 8], [10, 12, [14]]]
```

Marcus cannot find or recall the implementation of deep_map. Please help him by providing an implementation of deep_map.
[2m]

[4f] Towards the end of the project, Marcus suddenly fell sick, leaving you the only one to finish up the project. Implement the check_winner function which takes in a grid, and a to_win variable, representing the number of tokens in a row to win. The default of to_win should be 4. You may use the previously implemented functions to help you.
[4m]