**Project 9 Warmup**

Assume you have a 2d list called board that represents some sort of two-dimensional game board.

You want to figure out what squares are next to other squares. Assume you have variables row and col that are set to valid numbers for your board.

What square is to the right of board[row][col]? ____________________

What square is to the left of board[row][col]? ____________________

What square is above board[row][col]? ____________________

What square is below board[row][col]? ____________________

Write an if-statement that will be true if board[row][col] is on the top row of the board (and therefore has no upper neighbor):

Write an if statement that will be true if board[row][col] is on the bottom row of the board (and has no lower neighbor):

Write an if statement that will be true if board[row][col] is in the rightmost column of the board (and has no right neighbor):

Write an if statement that will be true if board[row][col] is in the leftmost column of the board (and has no left neighbor):

**Detecting identical neighbors**

Write a function has_identical_neighbor(board, row, col) that will return True if board[row][col] has at least one neighboring square that with the same number as board[row][col].

Hint: Use four if statements to check the four neighbors. You don't need any loops.

Example:

```
boardA = [[1, 3, 2, 1],
          [1, 4, 3, 3],
          [2, 4, 1, 3]]
```

`has_identical_neighbor(boardA, 0, 0)` should return `True` because the `1` in the upper left corner has another `1` below it.
`has_identical_neighbor(boardA, 2, 1)` should return `True` because the `4` in the bottom row has another `4` above it.
`has_identical_neighbor(boardA, 1, 2)` should return `True` because the `3` in the middle row has another `3` next to it.
`has_identical_neighbor(boardA, 2, 2)` should return `False` because the `1` in the bottom row has no other `1` next to it.

**"Spreading" a number around the board**

Write a function called spread(board, num) that searches a board for all -1s.  Any -1 is copied to its left/right/lower/upper neighbors, *as long as the neighbor matches the num parameter*.  Return the number of -1s added to the board (so return 0 if nothing was changed).

Imagine a board like this:

```
boardB = [[4, 3, 2, 1],
          [1, 4, 3, 3],
          [2, -1, 4, 4],
          [3, 4, 4, 1]]
```

We call `spread(boardB, 4)` which changes the board to this:

```
boardB = [[4, 3, 2, 1],
          [1, -1, 3, 3],
          [2, -1, -1, 4],
          [3, -1, 4, 1]]
```

The function call above will return `3`, meaning three 4's were changed to -1s. Notice how the -1 has spread to three additional squares. Then we call `spread(boardB, 4)` again. Now the board changes to:

```
boardB = [[4, 3, 2, 1],
          [1, -1, 3, 3],
          [2, -1, -1, -1],
          [3, -1, -1, 1]]
```

and the function call returns `2`, since changed two more 4s into -1s. Then we would `call spread(boardB, 4)` one more time, but the board wouldn't change, because there are no more 4s to change into -1s. Note that the 4 in the upper left corner doesn't change to -1 because it's not directly next to any of the -1s. So the function returns zero (and we can therefore stop calling it).

**Simulating gravity**

Write a function called gravity(board) that loops over the board and looks for a situation where there is a number in a square and a 0 in the square below it.  When this situation is found, copy the number into where the 0 was (simulating the number "falling down" the column) and replace the original number with a zero.  Return the number of numbers that were "lowered."

Note that because this function can only move a number one square down at a time, if a number needs to move 2 or more squares down a column (for instance, if there are multiple zeros back to back in a column) then you will need to call the function multiple times to get the numbers to fall all the way down.

Imagine a board like this:

```
boardC = [[1, 3, 2, 1],
         [1, 0, 0, 3],
         [2, 2, 0, 0],
         [0, 0, 0, 1]]
```

After calling `gravity(board)` **a few times in a row**, the board will look like:

```
boardC = [[0, 0, 0, 0],
         [1, 0, 0, 1],
         [1, 3, 0, 3],
         [2, 2, 2, 1]]
```