Author

[sir\_ementaler](https://codefights.com/profile/sir_ementaler)

https://codefights.com/img/coins_new.png3000

Determine results of the player's move in a simple puzzle video game.

The game is based on widely used, classic mechanics: it's played on a rectangular board filled with various symbols, represented in code by a matrix of positive numbers. Following some unspecified rules, a player rearranges some of the symbols to create a horizontal or vertical *line* of at least 3identical symbols in consecutive cells. Those symbols vanish, leaving empty cells. If there were any symbols above, they "fall down" until there's nowhere to fall anymore. This in consequence may produce additional *lines*, which again vanish, and the process repeats until no more *lines* can be found.

Given the state of the board immediately after the player's move, return the stable state it will reach after the described process.

**Example**

For

board = [[2, 3, 2, 1, 2],

[2, 1, 4, 4, 2],

[3, 3, 4, 4, 2],

[3, 3, 3, 3, 1],

[2, 3, 1, 1, 2]]

the output should be

SimulatePuzzleGame(board) = [[0, 0, 0, 0, 0],

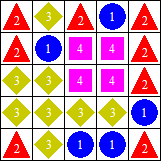
[2, 0, 0, 0, 0],

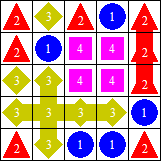
[2, 0, 2, 1, 0],

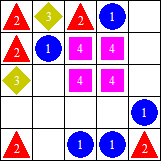
[3, 0, 4, 4, 1],

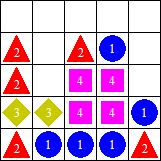
[2, 3, 4, 4, 2]]

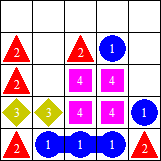
The example is illustrated below in a series of pictures:

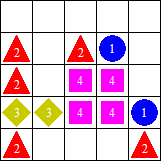
Initial board layout:  


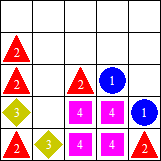
Three different *lines* are found, two of length 3 and one of length 4:  


Symbols located on the *lines* are removed:  


"Floating" symbols fall down:  


One more *line* of length 3 is found:  


Symbols located on the *line* vanish:  


"Floating" symbols fall down. There are no more *lines* so the process ends and returns the representation of the board.  


* **[input] array.array.integer board**

A rectangular matrix between 5 × 5 and 20×20 inclusive in size, representing the board. The matrix contains only numbers from the range [1,9] (no zeros). It is not guaranteed that any *line* is present.

* **[output] array.array.integer**

A matrix of the same dimensions as the input, representing the board after described operations.

<https://codefights.com/challenge/aTnxNZYkjg4KQwNXj>

**public** **class** Celda

{

**public** **int** Fila;

**public** **int** Columna;

**public** **int** IndiceColor;

**public** Celda()

    {

        IndiceColor = 0;

    }

**public** Celda(**int** fila, **int** columna, **int** indiceColor)

    {

**this**.Fila = fila;

**this**.Columna = columna;

**this**.IndiceColor = indiceColor;

    }

**public** **override** **bool** Equals(**object** obj)

    {

        Celda c = (Celda)obj;

**if** (**this**.Fila == c.Fila && **this**.Columna == c.Columna)

        {

**return** **true**;

        }

**return** **false**;

    }

}

**public**  **void** Sincronizar\_Queue( Celda[,] matriz,**int** filas, **int** cols)

{

*//BAJA TODAS LAS CELDAS SI HAY ESPACIOS ABAJO*

**for** (**int** col = 0; col < cols; col++)

    {

        Queue<Celda> pila = new Queue<Celda>();

**int** apila = 0; *//HAGO UNA COPIA SIN ESPACIOS EN UNA COLA => RECORRIENDO LA COLUMNA DESDE ABAJO HACIA ARRIBA*

**for** (**int** fila = filas - 1; fila >= 0; fila--)

        {

**if** (matriz[fila, col].IndiceColor != 0)

            {

                Celda c = matriz[fila, col];

                c.Fila = filas - 1 - apila; *// !! LE TENGO QUE PASAR LA NUEVA DIRECCION DE LA FILA !!!! (desde abajo hacia arriba, por eso => \_filas - 1 - apila;*

                pila.Enqueue(c);

                apila++;

            }

        }

*//COMPLETO LAS CELDAS QUE QUEDAN ARRIBA CON VALORES NULOS,*

*//(LAS VACIAS)*

**for** (**int** i = apila; i < filas; i++)

        {

            pila.Enqueue(new Celda());

        }

*//DESAPILO DESDE ABAJO HACIA ARRIBA,*

*//DESDE EL PRIMERO QUE INGRESÓ A LA COLA,*

*//POR ESO USO QUEUE EN LUGAR DE STACK*

**for** (**int** fila =  filas - 1; fila >= 0; fila--)

             matriz[fila, col] = pila.Dequeue(); *// listaAux[desapila];*

    }

}

**void** ponerEnCero(Celda[,] matriz, **int** filas, **int** cols)

{

    List<List<Celda>> filasAEliminar = new List<List<Celda>>();

*//recorro horizontalmente*

**for** (**int** i = 0; i < filas; i++)

    {

**int** j = 1;

**while** (j < cols)

        {

**int** len = 1;

            List<Celda> eliminar = new List<Celda>();

            eliminar.**Add**(matriz[i, j - 1]);

**while** (j < cols &&matriz[i, j - 1].IndiceColor!=0 && matriz[i, j - 1].IndiceColor == matriz[i, j].IndiceColor)

            {

                eliminar.**Add**(matriz[i, j]);

                len++;

                j++;

            }

**if** (len > 2)

            {

                filasAEliminar.**Add**(eliminar);

            }

            j++;

        }

    }

*//recorro verticalmente*

**for** (**int** j = 0; j < cols; j++)

    {

**int** i = 1;

**while** (i < filas)

        {

**int** len = 1;

            List<Celda> eliminar = new List<Celda>();

            eliminar.**Add**(matriz[i - 1, j]);

**while** (i < filas && matriz[i - 1, j].IndiceColor !=0 && matriz[i - 1, j].IndiceColor ==matriz[i, j].IndiceColor)

            {

                eliminar.**Add**(matriz[i, j]);

                len++;

                i++;

            }

**if** (len > 2)

            {

                filasAEliminar.**Add**(eliminar);

            }

            i++;

        }

    }

*//pongo en cero cada celda de la lista*

**foreach** (List<Celda> lista **in** filasAEliminar)

    {

**foreach** (Celda celda **in** lista)

        {

            matriz[celda.Fila, celda.Columna].IndiceColor = 0;

        }

    }

}

**void** mostrarMatriz(Celda[,] matriz, **int** filas, **int** cols)

{

**for** (**int** i = 0; i < filas; i++)

    {

**for** (**int** j = 0; j < cols; j++)

        {

            Console.Write(matriz[i, j].IndiceColor + " ");

        }

        Console.WriteLine();

    }

}

**bool** iguales(Celda[,] a, Celda[,] b, **int** filas, **int** cols)

{

**for** (**int** i = 0; i < filas; i++)

    {

**for** (**int** j = 0; j < cols; j++)

        {

**if** (a[i, j].IndiceColor != b[i, j].IndiceColor)

            {

**return** **false**;

            }

        }

    }

**return** **true**;

}

**void** copiar(Celda[,] dest, Celda[,] origen, **int** filas, **int** cols)

{

**for** (**int** i = 0; i < filas; i++)

    {

**for** (**int** j = 0; j < cols; j++)

        {

            dest[i, j] = origen[i, j];

        }

    }

}

**int**[][] SimulatePuzzleGame(**int**[][] board)

{

**int** filas =  board.Length;

**int** cols = board[0].Length;

    Celda[,] matriz = new Celda[filas, cols];

**for** (**int** i = 0; i < filas; i++)

    {

**for** (**int** j = 0; j < cols; j++)

        {

            matriz[i, j] =

               new Celda(i, j, board[i][j]);

        }

    }

    Celda[,] actual = new Celda[filas, cols];

    copiar(actual, matriz, filas, cols);

**while** (**true**)

    {

*//mostrarMatriz(matriz, filas, cols);*

        copiar(actual, matriz, filas, cols);

        ponerEnCero(matriz, filas, cols);

        Sincronizar\_Queue(matriz, filas, cols);

**if** (iguales(actual, matriz, filas, cols))

        {

**break**;

        }

    }

**int**[][] res = new **int**[filas][];

**for** (**int** i = 0; i < filas; i++)

    {

        res[i] = new **int**[cols];

**for** (**int** j = 0; j < cols; j++)

        {

            res[i][j] = matriz[i, j].IndiceColor;

        }

    }

**return** res;

*//  mostrarMatriz(matriz, filas, cols);*

}