Assignment 1 - Matrix

a) Create a method with signature:
   `void InitMatrix2D(int[,] matrix)`
   This method fills the (parameter) matrix with the numbers 1 until *n*
   where *n* is the number of elements of the matrix (see screenshot →).
   Use a nested for-loop.

```
 1  2  3  4  5  6  7  8
 9 10 11 12 13 14 15 16
17 18 19 20 21 22 23 24
25 26 27 28 29 30 31 32
33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48
49 50 51 52 53 54 55 56
57 58 59 60 61 62 63 64
```

b) Create a method with signature:
   `void DisplayMatrix(int[,] matrix)`
   This method displays the given array on the screen.
   Use a nested for-loop.
   → Test the methods (Init and Display, by calling them from the Start method) with arrays of different size (for example: 8x8, 11x11, 8x10).

c) Create an alternative Init method:
   `void InitMatrixLinear(int[,] matrix)`
   This method does exactly the same as method `InitMatrix2D` (fill the matrix), but now only 1 for-loop is used.
   This one loop has a loop-variable with values 1 until *n* (so, in the example above: 1 until 64). *You only need the (current) value to determine in which row and column it must be stored.*
   → Test the method with arrays of different size.

d) Create an alternative Display method:
   `void DisplayMatrixWithCross(int[,] matrix)`
   This method displays the numbers at one main diagonal red
   (foreground color) and the numbers at the other main diagonal
   yellow (background color). All other numbers are white.
   Use a nested for-loop, and use the loop-variables (row and column)
   to determine the color to use (with either
   `Console.ForegroundColor` or `Console.BackgroundColor`).
   The standard Console color can be reset with
   `Console.ResetColor()`.

```
  1   2   3   4   5   6   7   8   9  10  11
 12  13  14  15  16  17  18  19  20  21  22
 23  24  25  26  27  28  29  30  31  32  33
 34  35  36  37  38  39  40  41  42  43  44
 45  46  47  48  49  50  51  52  53  54  55
 56  57  58  59  60  61  62  63  64  65  66
 67  68  69  70  71  72  73  74  75  76  77
 78  79  80  81  82  83  84  85  86  87  88
 89  90  91  92  93  94  95  96  97  98  99
100 101 102 103 104 105 106 107 108 109 110
111 112 113 114 115 116 117 118 119 120 121
```

Assignment 2 – Search number

a) Create a method with signature:
      `void InitMatrixRandom(int[,] matrix, int min, int max)`
    This method fills the given array with random numbers between min and max.
    Use a nested for-loop.

```
94 49 50 21 33 40 42 84 30 31
 3 24  1 84 36 62 15 81 84  8
93 69 50 52 81 96 74 45 15 95
94 14 36 47 45 60 16 24  3 69
20 35 27 53 32 98 39 46 47  4
47 80 87 23 98 24 98 98 40 36
66 97 69 35 37 89 88 53 38 75
45 91 78 14 88 57 41 96 19 82
```

b) Copy method DisplayMatrix of the previous assignment.
    → Test the methods (Init and Display) by calling them from the Start method.

c) Now we are going to search numbers in the generated matrix. These numbers are stored at certain positions in
    the matrix. We can indicate/reference these positions with a row value and a column value (two values).
    However, a method can only return one value. If we create a class Position we can create a method that returns
    a position.
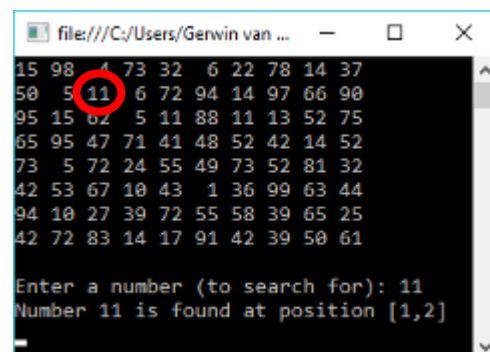   → Create a `class Position` with two `int` values: `row` and `column`.

d) Create a method with signature:
      `Position SearchNumber(int[,] matrix, int number)`
    In this method we will search the position (row and column) in the matrix where the (given) number is stored.
    Search the array <u>from top to bottom, from left to right</u> and stop searching at the first position where the
    number is stored.
    Return a Position containing the row and column of the position
    found.

    Let the user enter a number (in the `Start` method). Use method
    `SearchNumber` to find the entered number, and display the
    position.

```
file:///C:/Users/Gerwin van ...  —  □  ×
15 98  4 73 32  6 22 78 14 37
50  5 11  6 72 94 14 97 66 90
95 15 62  5 11 88 11 13 52 75
65 95 47 71 41 48 52 42 14 52
73  5 72 24 55 49 73 52 81 32
42 53 67 10 43  1 36 99 63 44
94 10 27 39 72 55 58 39 65 25
42 72 83 14 17 91 42 39 50 61

Enter a number (to search for): 11
Number 11 is found at position [1,2]
```

e) Also create a method with signature:
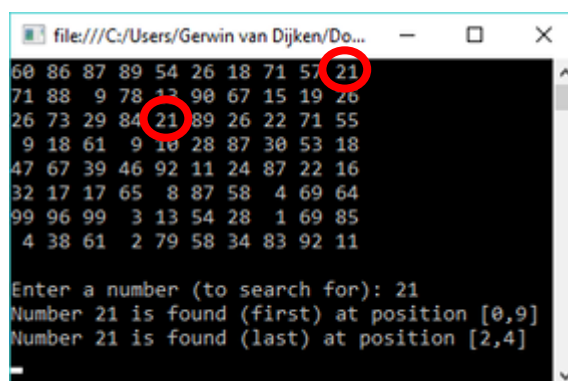   `Position SearchNumberBackwards(int[,] matrix, int number)`
    In this method we will search the position in the matrix where the (given) number is stored. Search the array
    <u>from bottom to top, from right to left</u> and stop searching at the 'first' position where the number is stored. Now
    the result will be the <u>last position</u> where the number is located.
    Return a Position containing the row and column of the position found.

    Call this method from the `Start` method and now display the first and the last position of the entered number.
    → Make sure to (also) test the program with a number in the first and last row and in the first and last column.
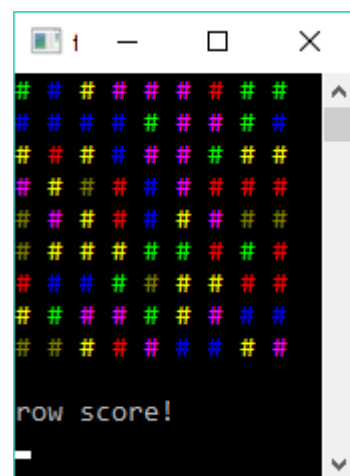
```
file:///C:/Users/Gerwin van Dijken/Do...  —  □  ×
60 86 87 89 54 26 18 71 57 21
71 88  9 78 13 90 67 15 19 26
26 73 29 84 21 89 26 22 71 55
 9 18 61  9 10 28 87 30 53 18
47 67 39 46 92 11 24 87 22 16
32 17 17 65  8 87 58  4 69 64
99 96 99  3 13 54 28  1 69 85
 4 38 61  2 79 58 34 83 92 11

Enter a number (to search for): 21
Number 21 is found (first) at position [0,9]
Number 21 is found (last) at position [2,4]
```

Assignment 3 – Candy Crush

a) Create an enum `RegularCandies` with the following options: JellyBean (red), Lozenge (orange), LemonDrop (yellow), Gum Square (green), LollipopHead (blue), Jujube Cluster (purple).

b) Create (in the `Start` method) a 2-dimensional array of type `RegularCandies` (a matrix in which regular candies can be stored). Give this array the name 'playingField'.

c) Create a method with signature:
    `void InitCandies(...)`
that receives the 2-dim array as a parameter. Fill the array with random candies (all candies can be used). You can use a conversion from an int-value to the corresponding enum-value.

d) Create a method with signature:
    `void DisplayCandies(...)`
that receives the 2-dim array as a parameter. This display-method displays each candy with a different color. The standard Console color can be reset with `Console.ResetColor()`.

e) Create a methode with signature:
    `bool ScoreRowPresent(...)`
that receives the 2-dim array as a parameter. This method returns true if there are 3 (or more) symbols next to each other in one row, otherwise it returns false. To determine '3 in a row' check all rows (one by one), and in each row remember the 'current candy' and set a counter to 1. If the next candy is the same (as the previous one) then increase the counter, otherwise reset the counter to 1 and set the 'current candy' again. If the counter becomes 3 then you can return true.
→ Check if there are (>=) 3 symbols adjacent in one row.

f) Also create a method with signature:
    `bool ScoreColumnPresent(...)`
This method returns true if there are 3 (or more) symbols above each other in one column, false otherwise.
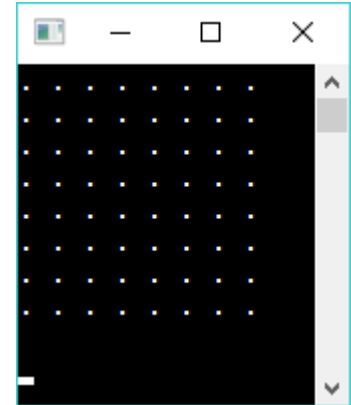→ Check if there are (>=) 3 symbols adjacent in one row or column.

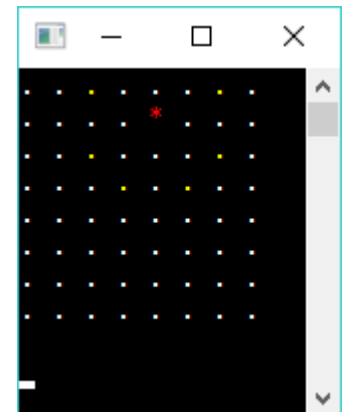## Assignment 4 – Knight movements *(not mandatory)*

a) Create a 2-dimensional int-array 'chessboard' with 8 rows and 8 columns.
    The possible values in the elements of this array are:
    0 = 'empty', 1 = 'occupied', 2 = 'possible move'.

b) Create a method with signature:
        void InitChessboard(int[,] chessboard)
    This method clears alle cells of the chessboard (value 0).

c) Create a method with signature:
        void DisplayChessboard(int[,] chessboard)
    This method displays the chessboard (0 = ⬛ 1 = ✳ 2 = ✳).
    → Test the (Init and Display) methods by calling them from the Start
    method.

d) Create a method with signature:
        Position PositionKnight(int[,] chessboard)
    This method positions a Knight (chess piece) at a random cell of the
    chessboard (value 1) and returns the position (use a class Position).
    → Test the method by calling it from the Start method, together with
    method DisplayChessboard.

e) Create a method with signature:
        void PossibleKnightMoves(int[,] chessboard, Position position)
    This method determines the possible positions (cells) where the Knight can
    move to (value 2). The current position of the Knight is at position.
    There are 8 possible positions. Do not hardcode these positions but use a
    loop (or multiple loops). Be aware that you can't always fill all 8 positions!
    → Test the method by calling it from the Start method.