Welcome to the Numbrix project. This is a fascinating application of using both two-dimensional arrays and recursion in order to solve complex number puzzles.

This is another "Pair Programming" Assignment

While you are working on this project, you will use the "Pair Programming" approach. In pair programming, two programmers share one computer. One student is the "driver," who controls the keyboard and mouse. The other is the "navigator," who observes, asks questions, suggests solutions, and thinks about slightly longer-term strategies. You should switch roles about every 20 minutes. For this assignment each partner will receive the same grade.

Be sure that both of you have duplicate saved copies of your work each day. You need to have something to work with tomorrow in the event your partner is absent.

---

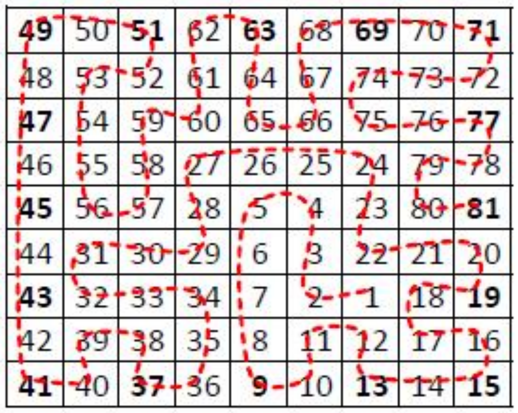
acknowledgements to Glen Martin for this assignment

For this project you will complete a program that solves Numbrix puzzles, which consist of a grid with numbers in some of the cells. The first puzzle supplied for this assignment is depicted on the right.

49		**51**		**63**		**69**		**71**
47								**77**
45								**81**
43								**19**
41		**37**		**9**		**13**		**15**

A solved Numbrix puzzle contains all the numbers from 1 to *rows* x *cols* (9 x 9 = 81 in this example) filled in. The original numbers must be unchanged, and consecutive numbers must be next to each other either vertically or horizontally.

This is what a completed solution to the first puzzle looks like. (The red dashed line shows how all of the numbers are connected.)



49	50	**51**	62	**63**	68	**69**	70	**71**
48	53	52	61	64	67	74	73	72
47	54	59	60	65	66	75	76	**77**
46	55	58	27	26	25	24	79	78
45	56	57	28	5	4	23	80	**81**
44	31	30	29	6	3	22	21	20
43	32	33	34	7	2	1	18	**19**
42	39	38	35	8	11	12	17	16
41	40	**37**	36	**9**	10	**13**	14	**15**

The most straightforward way to attack this problem is to try all the possible positions for the remaining numbers and see which ones solve the puzzle. This can be accomplished with a recursive depth-first search.

However, in this puzzle, there are 81 - 16 = 65 numbers to place. Therefore, there are 65! (65 factorial) possible placements for the remaining 65 numbers, only one of which is correct. 65! is a pretty large number. It's even greater than 10^{90}, a one followed by 90 zeros.

How long would it take to try 10^{90} different possible solutions? Let's assume that we have a smokin' fast program running on a really fast computer that can compute a quadrillion (1,000,000,000,000,000 = 10^{15}) possible number placements every second. Then it would take more than $10^{90} / 10^{15} = 10^{75}$ seconds for our program to complete. But how long is 10^{75} seconds?

10^{75}/ 60 / 60 / 24 / 365 is more than 10^{67} years. Since I'm pretty sure neither of us will be around by then, we need a faster algorithm.

The code you are going to write solves a Numbrix using a recursive depth-first search *with pruning*. "Pruning" means that your code will check each number before it is placed into the puzzle and if the number doesn't "fit", it will not explore that branch of the search any further. Using this strategy, a puzzle can be solved in seconds!

Our search for a solution begins with the **`solve`** method which iterates through each element (row **`r`** and column **`c`**) of **`grid`** and attempts to solve the puzzle by starting with a **`1`** in that location. It does this by calling **`recursiveSolve(r, c, 1)`** to attempt a solution beginning with a **`1`** at row **`r`** and column **`c`**.

`recursiveSolve` is the method that performs the recursive depth-first search. After placing a number **`n`**, it recursively attempts to place the number **`n+1`** in the location above, below, left, and right of the location where it placed **`n`**. It continues to attempt placing subsequent numbers until the last number is successfully placed.

Obviously, the program is unlikely to be successful on every placement attempt. When a number **`n`** can't be successfully placed at row **`r`** and column **`c`**, the **`recursiveSolve`** method returns and lets the method invocation that called it attempt a different placement. **`recursiveSolve`** returns when:

- either **`r`** or **`c`** is outside of **`grid`**. This is base case one.
- the current location contains `0` (empty), but the number to be placed was used elsewhere in the original puzzle. This is base case two and the first "pruning" situation.
- the current location contains a number, but it's not equal to the number to be placed. This is base case three and the second "pruning" situation.
- the puzzle is solved. In this case we print the solution and return to look for more solutions. This is base case four.
- the four recursion calls have been completed.

1. The starter code has been provided for you. It contains two java files and six puzzle data files:

    a. `NumbrixMain` is the application class for this project. This class has been completely written for you. There is no need to make any additions or changes to it.
    b. `Numbrix` objects represent a Numbrix puzzle. This is where you will place your code.
    c. Each puzzle data file (there are six of them) is a text file of integers. The first two values are the number of rows followed by the number of columns in the grid. The numbers that follow those are the values for each cell (in row-major) order. A zero represents a blank (unsolved) cell in the grid. You may assume that there are always row *x* column + 2 integers in any puzzle file for this game, but do not assume that every puzzle is square (where the number of rows equals the number of columns).

2. The **`Numbrix`** class already contains two instance variables: **`grid`** and **`used`**. **`grid`** will contain the puzzle and **`used`** will indicate if a number was used in the original puzzle. These are the only instance variables that are needed.

3. First, complete the constructor for the **`Numbrix`** class. It should do the following:

   a. Create a **`Scanner`** object to input data from the file named in **`filename`**. Use **`new Scanner(new File(fileName))`** to instantiate your **`Scanner`**.
   b. Read in the number of rows and columns in the puzzle (use the **`Scanner`** method **`nextInt()`**). These are the first and second numbers in the data file respectively.
   c. Instantiate the **`grid`** 2D array to have the input number of rows and columns.
   d. Instantiate the **`used`** array to have *rows * columns + 1* elements. The extra element allows you to use the numbers in the puzzle as indices. Otherwise you would need to subtract **`1`** to prevent an **`ArrayOutOfBoundsException`**. The element at index 0 is ignored.
   e. Input the puzzle numbers and use them to fill in **`grid`** and **`used`**. **`used[num]`** should be **`true`** if **`num`** is in the original input puzzle. It should be **`false`** otherwise.

4. Complete the **`toString`** method to return a **`String`** with the **`grid`** data. The requirements for the string are as follows:

   a. The data must be in row-major order.
   b. 0s must be indicated by dash (minus sign) characters.
   c. There must be one tab character after each number or dash (**`"\t"`**).
   d. There must be one new line character after each row.
   e. There must be no extraneous spaces of other characters.

5. Test your program and correct any errors. It should produce the following output for the first puzzle (tab width may vary):

```
Puzzle #1 ****************************
49	-	51	-	63	-	69	-	71
-	-	-	-	-	-	-	-	-
47	-	-	-	-	-	-	-	77
-	-	-	-	-	-	-	-	-
45	-	-	-	-	-	-	-	81
-	-	-	-	-	-	-	-	-
43	-	-	-	-	-	-	-	19
-	-	-	-	-	-	-	-	-
41	-	37	-	9	-	13	-	15
```

   Note: if you don't see the first puzzle print out on the screen (but you see the rest of them), then when you are in the Terminal Window, select Options, then Unlimited Buffering, and then run your code again.

6. Complete the **`solve`** method. This method should attempt to solve the puzzle by starting with a **`1`** in each available **`grid`** element. For each element of **`grid`**, solve should call **`recursiveSolve(r, c, 1)`** where **`r`** and **`c`** are the row and column of the **`grid`** location in which to attempt to place the **`1`**.

7. Test your new **`solve`** method by adding temporary code to **`recursiveSolve`** that prints **`r`**, **`c`**, and **`n`**. Make sure that **`solve`** calls **`recursiveSolve`** for all possible **`r`** and **`c`**. Also make sure that **`n`** is always **`1`**. Then, when you are satisfied that **`solve`** is working correctly, remove the temporary code from **`recursiveSolve`**.

8. Complete the **`recursiveSolve`** method. This is the recursive method that performs the depth-first search for solutions. *Follow the instructions carefully to complete this method.* Complete it as follows:

   a. Make sure that **`r`** and **`c`** specify an element inside the **`grid`**. If **`r`** is an invalid row index, or **`c`** is an invalid column index, then **`return`**. This is base case one.

   b. Create and initialize a **`boolean`** variable named **`zero`**. This variable needs to be true iff (if and only if) **`grid[r][c]`** contains a `0`. Do this with one statement of the following form: **`boolean zero = ... ;`**

   c. The next two base cases (d) and (e) consist of situations where it does not make sense to place **`n`** in **`grid[r][c]`**. They constitute the pruning discussed earlier.

   d. If **`zero`** is **`true`**, but **`n`** is in the original puzzle, then **`n`** can't be placed in **`grid[r][c]`** because it is already used elsewhere. In this case, your method should **`return`**. Utilize **`used`** to determine if **`n`** is in the original puzzle. This is base case two.

   e. If **`zero`** is **`false`** and **`grid[r][c]`** contains a number other than **`n`**, then your method should **`return`**. This is base case three.

   f. At this point, it makes sense to store **`n`** in **`grid[r][c]`**. Go ahead and do that.

   g. Now check to see if the puzzle is solved. See if **`n`** equals the product of the number of rows and columns in **`grid`**. If so, you should print **`this`** with **`System.out.println(this)`**; which in turn calls **`toString`** implicitly. This is the fourth base case. Don't return yet, though, because we want to also check for other solutions.

   h. If the puzzle isn't solved then make four recursive calls. There four calls should specify the element that is up, down, left, and right of the current element. The 3rd parameter should be **`n + 1`**.

   i. Finally, if **`zero`** is **`true`**, then set **`grid[r][c]`** back to `0` (which will allow us to check for other solutions).

9. Test your program and correct any errors. It should produce the following output for the first puzzle (tab width may vary):

```
Puzzle #1 ****************************
49    50    51    62    63    68    69    70    71
48    53    52    61    64    67    74    73    72
47    54    59    60    65    66    75    76    77
46    55    58    27    26    25    24    79    78
45    56    57    28    5     4     23    80    81
44    31    30    29    6     3     22    21    20
43    32    33    34    7     2     1     18    19
42    39    38    35    8     11    12    17    16
41    40    37    36    9     10    13    14    15
```

10. Then, make sure your program is able to solve all six of the puzzles that were provided. Did you notice anything unusual with the solution for the last game?

Hope you have as much fun with this project as I did! When you are finished with your work, demonstrate it for me to see.