Objectives

1. Solve problems using recursion

Setup

1. Download the starter code from mrmaycs.com
2. Extract the Lab16 folder into your H:/APCS/Lab8.4 folder
3. Use DrJava to write your code

Assignment

Write a recursive method that solves Numbrix puzzles designed by Marilyn vos Savan in Parade magazine. This lab was originally written by Robert Glen Martin at the school for the Talented and Gifted.

In this lab you will write a program that solves Numbrix puzzles. Numbrix puzzles were invented by Marilyn vos Savan, the author of the "Ask Marilyn" column in Parade Magazine. Numbrix puzzles can be found at http://www.parade.com/numbrix and various other online sites.

Numbrix puzzles consist of a grid with numbers in some of the cells. The 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**

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**

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.

The solution to this puzzle is depicted on the left.

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.

There are two java files in the starter code. These files are not in a project, so you should use file→open to to open them.

a. **`Runner`** – is the application class for this project. **This class has been completely written for you. Do not make any additions or changes to it.**
b. **`Numbrix`** – objects of this class represent a Numbrix puzzle. **This is where you will place your code.**

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 needed. **Do not add any additional instance variables.**

Part 1

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

- Instantiate the **`grid`** 2D array to have the input number of rows and columns.
- 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. (see more detail about `used` below)
- 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.
  Use a nested loop that iterates over **`grid`** in row-major order. Inside the inner loop, use **`s.nextInt()`** to read the *next* value from the file `Data.txt`.

The Used Array

The `used` array is a 1d array that keeps track of which values have already been *used* in the puzzle.

Consider a Numbrix puzzle that took place on a 3 x 4 grid and contained the values 1 – 12.

		9	
2			11
	4		

Each index of the `used` array represents one of the values to be placed in the Numbrix puzzle. Since the Numbrix puzzle begins with the number 1, you will ignore the value at index 0 of the `used` array. Since the last value to be placed in the Numbrix puzzle is 12 the `used` array should have 13 elements (so that the last index is 12)

Each number that is already in the used array should be set to `true` in the `used` array.

`boolean[] used =`

0	1	**2**	3	**4**	5	6	7	8	**9**	10	**11**	12
F	F	**T**	F	**T**	F	F	F	F	**T**	F	**T**	F

When you add the value 1 to the Numbrix puzzle, you should assign the value `true` to `used[1]`, when you add the value 8 to the Numbrix puzzle, you should assign the value `true` to `used[8]`. When you add the value `n` to the Numbrix puzzle, you should assign the value `true` to `used[n]`.

Similarly, if you *remove* the value `n` from the Numbrix puzzle, you should assign the value *`false`* to `used[n]`.

If you need to check whether or not a number has already been placed in the Numbrix puzzle you can use the following code:

```
if(used[n] == true)
{
  //the value n has already been placed somewhere in this puzzle!
}
```

Part 2

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

- The data must be in row-major order.
- 0s must be indicated by dash (minus sign) characters.
- There must be one tab character after each number or dash. Use the escape character **`\t`** to represent a tab.
- There must be one new line character after each row. Use the escape character **`\n`** to represent a new line.
- There must be no extraneous spaces or other characters.

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

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

Part 3

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**.

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**.

When you are satisfied that **`solve`** is working correctly, remove the temporary code from **`recursiveSolve`**.

Part 4

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 **`grid[r][c]`** contains a 0. **Do this with <u>one</u> statement of the following form: `boolean zero = …;`**

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

c. 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*.
d. If **`zero`** is **`false`** and **`grid[r][c]`** contains a number other than **n**, then your method should return. *This is base case three.*
e. At this point, it makes sense to store **n** in **`grid[r][c]`**. Go ahead and do that.
f. 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 because we need to remove the number at **`grid[r][c]`** and look for other solutions.
g. 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`**.
h. Finally, if **`zero`** is **`true`**, then set **`grid[r][c]`** back to **0**.

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

```
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
```

One final note about Numbrix puzzles and this program: A well-constructed Numbrix puzzle will only have one solution. However, in the case where there is more than one solution, your program should find and print all of them.