3. **A Sudoku solver** (30 points)

In this problem, you will write a program that solves Sudoku puzzles using recursive backtracking.

A Sudoku puzzle consists of a 9x9 grid in which some of the cells have been filled with integers from the range 1-9. To solve the puzzle, you fill in the remaining cells with integers from the same range, such that each number appears exactly once in each row, column, and 3x3 subgrid. The left-hand image below shows an example of an initial puzzle, and the right-hand image shows its solution.

	4				3	7		
	8	9	7			1		
					4	2		
								1
6			5	1	8			9
2								
		5	3					
		6			1	9	4	
		1	2				6	

1	4	2	9	8	3	7	5	6
5	8	9	7	2	6	1	3	4
7	6	3	1	5	4	2	9	8
9	5	8	4	3	2	6	7	1
6	3	7	5	1	8	4	2	9
2	1	4	6	9	7	5	8	3
4	7	5	3	6	9	8	1	2
3	2	6	8	7	1	9	4	5
8	9	1	2	4	5	3	6	7

Most of the functionality of your program should go in a class called `Puzzle`, which you will use to represent an individual Sudoku puzzle. We have provided you with skeleton code for this class in the file `Puzzle.java`, including:

- a field called `values` that refers to a two-dimensional array of integers. This array is used to store the current contents of the cells of the puzzle, such that `values[r][c]` stores the current value in the cell at row r, column c of the puzzle. A value of 0 is used to indicate a blank cell.
- a field called `valIsFixed` that refers to a two-dimensional array of booleans. It is used to record whether the value in a given cell is fixed (i.e., part of the original puzzle). `valIsFixed[r][c]` is `true` if the value in the cell at row r, column c is fixed, and `valIsFixed[r][c]` is false if the value in that cell is not fixed. For example, in the original puzzle above, there is a fixed 4 in the cell at row 0, column 1 (the second cell in the top row), and thus `valIsFixed[0][1]` would be true for that puzzle.
- fields called `rowHasValue` and `colHasValue` that each refer to a two-dimensional array of booleans. These arrays allow us to determine if a given

row or column already contains a given value. For example, `rowHasValue[3][4]` will be true if row 3 already has a 4 in one of its cells.

- partial implementations of methods called `placeValue()` and `removeValue()` that place a value in a given cell and remove a value from a given cell by updating the fields mentioned above. You will need to add code to these methods to update any other fields that you add.
- a full implementation of a method called `readFrom()` that takes a `Scanner` as its only parameter and reads in a specification of a puzzle from that `Scanner`. This method assumes that a puzzle specification consists of nine lines of text – one for each row of the puzzle – and that each line contains nine digits separated by whitespace. Here again, 0 is used to indicate a blank cell. For example, the specification of the initial puzzle above would begin:

  ```
  0 4 0 0 0 3 7 0 0
  0 8 9 7 0 0 1 0 0
  ...
  ```

  The method reads in the puzzle specification and makes the corresponding changes to the fields mentioned above. You should **not** need to change this method, because it calls `placeValue()`, and you are already modifying that method as needed. However, we do recommend that you read this method and understand it.
- the full implementation of a method called `display()` that prints out the current state of the `Puzzle` object on which it is invoked.
- the skeleton of a method called `solve()` that you will implement. This is the recursive-backtracking method, and it should return `true` if a solution to the puzzle is found and `false` if no solution has been found (i.e., if the method is backtracking). If the initial call to this method returns false, that means that no solution can be found – i.e., that the initial puzzle is not valid. If there is more than one solution (which can happen if the initial puzzle does not have enough numbers specified), your code should stop after finding one of them.

  Each invocation of the `solve()` method is responsible for finding the value of a single cell of the puzzle. The method takes a parameter n, which is the number of the cell that the current invocation of this method is responsible for. We recommend that you consider the cells one row at a time, from top to bottom and left to right, which means that they would be numbered as follows:

  ```
  0  1  2  3  4  5  6  7  8
  9 10 11 12 13 14 15 16 1
  18 ...
  ```

- a partial implementation of a constructor. You will need to add code to initialize any fields that you add.

In addition to completing the methods mentioned above, you should also add to the `Puzzle` class any other fields or methods that are needed to maintain the state of a Sudoku puzzle and to solve it using recursive backtracking. **In particular, we recommend adding one or more fields to keep track of whether a given subgrid of the puzzle (i.e., a given 3 x 3 region) already contains a given value.**

We have also given you a separate file (`Sudoku.java`) that contains the `main()` method for your Sudoku solver. You shouldn't need to change this method (or any other code in this file), but we encourage you to review its use of the methods of the `Puzzle` class. In particular, note that it displays the puzzle after the solution is found, and thus it isn't necessary for your recursive-backtracking method to display it. Finally, we have given you four sample puzzle files: `puzzle1.txt`, `puzzle2.txt`, `no_solution.txt` (which has no solution), and `multi_sol.txt` (which has multiple solutions).

Additional hints and suggestions:

- Take advantage of the second template in the notes for recursive backtracking – the one in which the method returns a boolean.
- As mentioned above, the recursive `solve` method takes a single parameter `n` that represents the number of the cell that the current invocation is responsible for. You will need to use n to compute the row and column indices of the cell, and you should be able to do so using simple arithmetic operations (+, –, *, /, and %).
- Make sure that you take advantage of the `rowHasValue` and `colHasValue` fields – along with the field(s) that you will add to keep track of the values in a given subgrid of the puzzle – when deciding whether a particular value can be assigned to a particular cell. You should *not* need to scan through the puzzle to determine if a given value is valid. See the notes for n-Queens for another example of efficient constraint checking.
- Make sure that you use the `addValue()` and `removeValue()` methods when updating the state of the puzzle, and that you add code to these methods as needed to update any fields that you add to the `Puzzle` class.
- Make sure that you don't attempt to assign a new number to cells that are filled in the original puzzle – i.e., cells whose values are fixed. Your `solve()` method will need to skip over these cells somehow.

A sample run of the program is shown below:

```
Please enter the name of puzzle file: puzzle2.txt

Here is the initial puzzle:
-----------------------------------
|   | 4 |   |   |   | 3 | 7 |   |   |
-----------------------------------
|   | 8 | 9 | 7 |   |   | 1 |   |   |
-----------------------------------
|   |   |   |   |   |   | 4 | 2 |   |   |
-----------------------------------
|   |   |   |   |   |   |   |   |   | 1 |
-----------------------------------
| 6 |   |   | 5 | 1 | 8 |   |   |   | 9 |
-----------------------------------
| 2 |   |   |   |   |   |   |   |   |   |
-----------------------------------
|   |   | 5 | 3 |   |   |   |   |   |   |
-----------------------------------
|   |   |   | 6 |   |   |   | 1 | 9 | 4 |   |   |
-----------------------------------
|   |   |   | 1 | 2 |   |   |   |   | 6 |   |   |
-----------------------------------

Here is the solution:
-----------------------------------
| 1 | 4 | 2 | 9 | 8 | 3 | 7 | 5 | 6 |
-----------------------------------
| 5 | 8 | 9 | 7 | 2 | 6 | 1 | 3 | 4 |
-----------------------------------
| 7 | 6 | 3 | 1 | 5 | 4 | 2 | 9 | 8 |
-----------------------------------
| 9 | 5 | 8 | 4 | 3 | 2 | 6 | 7 | 1 |
-----------------------------------
| 6 | 3 | 7 | 5 | 1 | 8 | 4 | 2 | 9 |
-----------------------------------
| 2 | 1 | 4 | 6 | 9 | 7 | 5 | 8 | 3 |
-----------------------------------
| 4 | 7 | 5 | 3 | 6 | 9 | 8 | 1 | 2 |
-----------------------------------
| 3 | 2 | 6 | 8 | 7 | 1 | 9 | 4 | 5 |
-----------------------------------
| 8 | 9 | 1 | 2 | 4 | 5 | 3 | 6 | 7 |
-----------------------------------
```