CSE143 Section #18 Problems

The remaining problems involve writing code that involves two-dimensional arrays. The specific program we will consider is a Sudoku solver that uses recursive backtracking. As with the 8 queens problem, we will define a class that handles much of the low-level detail of the puzzle. In particular, we will have a class called Grid that keeps track of the current state of the 9x9 Sudoku grid. The basic class structure will be:

```
    public class Grid {
        public static final int SIZE = 9;
        private int[][] grid;

        // methods
    }
```

We are writing a fairly specific class with a size of 9, but it is best to use a named constant when possible to add to readability. The grid will store values between 1 and 9. We will assume that the value 0 means that the cell is empty (no value has been assigned to it).

Several of the methods involved use a cell number to refer to a particular location. The idea is that the cells are numbered starting at 0 in row/major order. In other words, we number all of the first row, then all of the second row, and so on. So the cell numbers are:

```
         0  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
        ...
        72 73 74 75 76 77 78 79 80
```

Because we are starting at 0, the highest cell number is 80.

7. Write a method called print that prints the grid contents to System.out with each row printed on a separate line and with a space after each grid value. Empty cells (cells that store a value of 0) should be printed as a dash. For example, if you have a Grid variable g and you make this call:

```
        g.print();
```

   you should get output that looks something like this:

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

```
     - 2 - - 1 5 - - 7
     4 - - 6 - - - 1 -
```

Instead of using the SIZE constant, write this method so that it would work no matter what the dimensions of the grid are and even if the rows had different numbers of elements in them.

8. Write a constructor for the Grid class that takes a Scanner as a parameter. The Scanner will contain values like those produced by print. In other words, cells that are occupied will be listed with a number between 1 and 9 and cells that are empty will be listed with some other token like a dash. You may assume the Scanner is open and that it contains a legal sequence of tokens (81 of them).

9. Write a method called place that takes a cell number and a value n and that stores the value n in that cell. Your method will have to convert the cell number to a row and column in your grid.

10. Write a method called remove that takes a cell number and that removes the value in that cell (resetting it to 0). As with the place method, you will have to convert the cell number to a row and column in your grid.

11. Write a method called getUnassignedLocation. It should look through the grid in row/major order and return the cell number of the first unoccupied cell. If there are no unoccupied cells, it should return the value -1.

12. Write a method called noConflicts that takes a cell number and a value n and that returns true if it is possible to store n in that cell without generating any Sudoku conflicts. Remember that Sudoku does not allow repetition in any given row, column, or block.

13. Write the recursive backtracking code that will search all possible solutions to the Sudoku puzzle. You should write a method called explore that takes a Grid as a parameter and that attempts to fill the grid with a solution. It should return whether or not a solution is found. Below is a method that calls the explore method and reports the result:

```
public static void solve(Grid g) {
    if (!explore(g))
        System.out.println("No solution.");
    else {
        System.out.println("One solution is as follows:");
        g.print();
    }
}
```

The Grid object has the following public member functions available to you:

```
getUnassignedLocation()     returns the cell number of the first
                            unoccupied cell (-1 if no such cell exists)
```

`noConflicts(cellNumber, n)`	returns whether it is legal to place the value n in the given cell without violating Sudoku constraints
`place(cellNumber, n)`	places the value n in the given cell
`remove(cellNumber)`	removes the value in the given cell, resetting it to be unoccupied