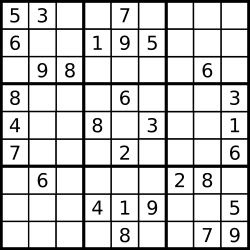
**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

[***Sudoku***](https://en.wikipedia.org/wiki/Sudoku) is a simple number puzzle that uses a 9x9 array of numbers as shown below. A correct solution to the puzzle requires the player to fill in all the empty squares with digits from 1-9 such that every row contains all digits from 1-9, every column has the digits 1-9 and the nine 3x3 subblocks each have the digits 1-9. Your task is to write a program that will solve incomplete Sudoku puzzles that will be read from files. This assignment requires you to use recursion in game environment to support [**backtracking**](https://en.wikipedia.org/wiki/Backtracking). **You must use recursion in your solution.**



# **What Is A Backtracking Algorithm?**

In **Backtracking Algorithms**, you try to build a solution one step at a time. If at some step it is clear that the current path that you are on cannot lead to a solution, you return back to the previous step (backtrack) and choose a different path.

Think of a maze - how do you find a way from an entrance to an exit? Once you reach a dead end, you must backtrack. But backtrack to where? - to the previous choice point. Backtracking is also known as *depth-first-search*.

Let's take a situation. Suppose you are standing in front of three tunnels, one of which is having a bag of gold at its end, but you don't know which one. So you'll try all three. First go in tunnel 1, if that is not the one, then come out of it, and go into tunnel 2, and again if that is not the one, come out of it and go into tunnel 3. So basically in backtracking we attempt solving a subproblem, and if we don't reach the desired solution, then undo whatever we did for solving that subproblem, and try solving another subproblem.

# **Approach**

You should use a *brute force* approach, which means that you won’t try to make your program smart, just persistent! In other words it will simply try every possible digit in each empty square until it finds one that does not violate the row, column, and subblock rules. As soon as it finds one that works, it goes to the next empty square and tries to put a digit there and continues in this way until the boxes are all filled or it finds an empty box that cannot be filled with any digit without violating the rules. When that happens, your program must *backup*, or return to a previous square and “undo” the last digit it placed there and try another. This process continues until the puzzle is solved or the program discovers that there is no solution.

## **Algorithm**

In order to make this work, your overall approach has to be transformed into a rigorous algorithm that organizes the trial-and-error process: you need to “know” where you are at all times and what digits you have and have not tried in any given square. It’s pretty straightforward to just traverse the squares of the puzzle by row (or by column) looking for an empty square. When you find one, you try all the digits from 1 to 9 (this can be inside a *for* or *while* loop). If you find a digit that can go there, you put it into the square, assume that it is the correct digit and recur to look at the “next” blank square. You’ll surely reach a point where the choices you’ve made so far are not correct and all numbers will be incorrect for a given empty square. So, you return failure to the caller of this invocation. The caller will then try the next number in its sequence, if there is one, which will cause another recursion. If the caller just finished its loop (it tried a 9 that failed), it will return failure to *its* caller, and so forth. Eventually, (if the board is a valid board) a correct number can be placed on the last available blank and that invocation sees that there are no more blanks left and returns success, which ripples all the way back up the recursion stack.

A recursive strategy for finding a solution to a Sudoku problem is as follows:

| **Base case 1** : If you are past the end of the board, then all previous locations must have been filled by previous recursive calls, and the puzzle is solved.  **Recursion case 1** : If this location already has a value (from the board file), it should not be changed. Recursively call for the next location.  **Recursion case 2 :** If this location did not already have a value, try the next legal value, then recursively attempt to find a solution from the next cell. If the recursive attempt returns no solution, try the next legal value and call recursively again.  If a solution is found for at least one legal value, return that a solution was found.  If no solution is found for all values for this cell, reset this cell to be empty, and return that there was not a solution found. |
| --- |

Every time the code recursively attempts to find a solution, it will fill cells in the Sudoku grid. If an attempt is not successful, before returning, your code must restore the Sudoku grid to have empty spaces in the correct places.

# **Program and Starter Code**

You will need to create the following files:

1. **Location.java**
2. **Board.java**
3. **Sudoku.java**

along with some data sets. Your main method should be in **Sudoku.java**. Starter code ([public/6P](http://cs.unh.edu/~cs416/public/6P)) includes several test files containing Sudoku boards.

## 

## **Location.java**

This class represents a location on a Sudoku board. It should store a row and column value, and include methods for getting the "next" location from the current one.

**public Location( int r, int c )**

The constructor for Location. It should store the row and column values.

**public int getRow()**

Returns the Location's row value.

**public int getColumn()**

Returns the Location's column value.

**public Location next()**

Gets the next Location after this one. Should proceed through a row by incrementing the column value, until it needs to move to the next row. For example, the next location from 1,1 is 1,2 but the next location from 1,8 is 2,0. Should return null if the next location would be outside the space of a standard Sudoku board (9 rows by 9 columns.)

**public String toString()**

Returns a string representation of this Location. It should be in this format: **row, col**

## 

## **Board.java**

This class represents the 9x9 board of values for the Sudoku puzzle. It should store the values for the current state of the puzzle, and contains methods to read in a board as well as methods to help check whether a value is allowed in a given row, column, 3x3 box, or location.

**public Board( Scanner sc )**

Constructor for the Board class. Creates the instance variables to store the values for a 9x9 board. Should use **readBoard** to do most of its work.

**public static int[][] readBoard( Scanner sc )**

Uses the provided Scanner to read the board and return a 2D integer array with the appropriate values. Input should be 9 rows of values with 9 characters each. These characters will be 1-9 to indicate a fixed number in the starting board, or a - (dash) to indicate an empty location. A value of 0 indicates an empty space in the returned array. Should return null if the board was invalid (contains invalid characters or does not contain enough values.) Example input:

| **-76-89-54**  **8-43-271-**  **-1--4683-**  **7-8-93-45**  **23-81-967**  **9-167-3-8**  **-834-759-**  **-5-9-16--**  **1--56-47-** |
| --- |

**public int get( int row, int col )**

Returns the value at row, col in the board.

**public void set( int row, int col, int value )**

Sets the value at row, col in the board. Does not need to confirm that the value is allowed, as these checks should be done before calling this method.

**public boolean containsInRow( int row, int number )**

Returns true if number was already contained in row, false otherwise.

**public boolean containsInCol( int col, int number )**

Returns true if number was already contained in col, false otherwise.

**public boolean containsInBox( int row, int col, int number )**

Returns true if number was already contained in the 3x3 box containing row, col, false otherwise. For instance, for 0,1 the method should check the box 0,0 through 2,2

**public boolean isAllowed(int row, int col, int number)**

Returns true if number would be valid to assign to row, col, false otherwise. For number to be allowed it should satisfy all of the rules for row, column, and 3x3 box. It should also return false if the location has a fixed value (one given from the input file.)

**public String toString()**

Returns a string representation of the board. Uses dashes to indicate empty spaces, and indicates the 3x3 boxes within the board. For example:

| **+-------+-------+-------+**  **| - 7 6 | - 8 9 | - 5 4 |**  **| 8 - 4 | 3 - 2 | 7 1 - |**  **| - 1 - | - 4 6 | 8 3 - |**  **+-------+-------+-------+**  **| 7 - 8 | - 9 3 | - 4 5 |**  **| 2 3 - | 8 1 - | 9 6 7 |**  **| 9 - 1 | 6 7 - | 3 - 8 |**  **+-------+-------+-------+**  **| - 8 3 | 4 - 7 | 5 9 - |**  **| - 5 - | 9 - 1 | 6 - - |**  **| 1 - - | 5 6 - | 4 7 - |**  **+-------+-------+-------+** |
| --- |

## 

## **Sudoku.java**

This class does the work of solving the Sudoku puzzle, as well as containing the main method to provide a text-based user interface. Stores a Board object which it uses in solving. It should also track statistics about the solving process: the number of recursive calls made, and the number of "backups" that had to be done.

**public Sudoku( Scanner sc )**

Constructor for the Sudoku class. Initializes the board and other instance variables.

**public boolean solve( Location loc )**

Recursively solves the board from the current configuration, using the backtracking algorithm described above. Returns true if a solution was found from the current configuration, false otherwise. Should only assign valid values to board locations, and should not overwrite locations that had values in the original board configuration.

**public int getRecursionCount()**

Returns the recursion count after the puzzle has been solved (or discovered to be unsolvable.)

**public int getBackupCount()**

Returns the backup count after the puzzle has been solved (or discovered to be unsolvable.)

**public Board getBoard()**

Returns the Sudoku board.

**public static void main( String[] args )**

The main method. Prompts the user for a file name, [creates a Scanner using that file](#sxyaom6cnrjj), creates a Sudoku object, prints the initial board, calls the solve method, and then prints the results, including the recursion count and backup count. Transcript below:

| **Enter the path to the sudoku file:**  **test1.txt**  **Initial configuration of the sudoku**  **+-------+-------+-------+**  **| - 7 6 | - 8 9 | - - 4 |**  **| - - 4 | 3 - 2 | - 1 - |**  **| - 1 - | - - 6 | 8 3 - |**  **+-------+-------+-------+**  **| 7 - 8 | - 9 3 | - - 5 |**  **| 2 3 - | - - - | - 6 7 |**  **| 9 - - | 6 7 - | 3 - 8 |**  **+-------+-------+-------+**  **| - 8 3 | 4 - - | - 9 - |**  **| - 5 - | 9 - 1 | 6 - - |**  **| 1 - - | 5 6 - | 4 7 - |**  **+-------+-------+-------+**  **Successful!**  **Final configuration of the sudoku**  **+-------+-------+-------+**  **| 3 7 6 | 1 8 9 | 2 5 4 |**  **| 8 9 4 | 3 5 2 | 7 1 6 |**  **| 5 1 2 | 7 4 6 | 8 3 9 |**  **+-------+-------+-------+**  **| 7 6 8 | 2 9 3 | 1 4 5 |**  **| 2 3 5 | 8 1 4 | 9 6 7 |**  **| 9 4 1 | 6 7 5 | 3 2 8 |**  **+-------+-------+-------+**  **| 6 8 3 | 4 2 7 | 5 9 1 |**  **| 4 5 7 | 9 3 1 | 6 8 2 |**  **| 1 2 9 | 5 6 8 | 4 7 3 |**  **+-------+-------+-------+**  **Recursion count = 94**  **Backup count = 7** |
| --- |

**Outline of how to create a Scanner from a file in Java:**

// get the file name from a Scanner built on System.in

File file = new File(fileName);

try {

Scanner boardScanner = new Scanner(file);

// Sudoku creation and solving go here

}

catch (FileNotFoundException e) {

System.out.println("File was not found.");

}