**Introduction to Sudoku**

Sudoku is a logic-based, number-placement puzzle. The objective is to fill a 9×9 grid of cells with digits so that each column, each row, and each of the nine 3×3 boxes that compose the grid contains all of the digits from 1 to 9. The puzzle creator provides a partially completed grid, which, for a well-posed puzzle, has a single solution. The same single digit may not appear twice in the same row, nor twice in the same column, nor twice in the same 3x3 box. -Wikipedia

Here is an example. The left Sudoku board shows the initial puzzle, with some of the cells specified and others left blank. The puzzle solver’s task is to fill in the blank cells, according to the Sudoku rules, and arrive at the solution shown on the right. (Verify for yourself that no row, column, or 3x3 box contains a duplicated digit.) Some Sudoku puzzles are easy; others are very, very hard.

|  |  |
| --- | --- |
| A typical Sudoku puzzle, with nine rows and nine columns that intersect at square spaces. Some of the cells are filled with a number; others are blank cells to be solved. | The previous puzzle, showing its solution. |

**Project Description**

For this coding class you will be writing a computer program to solve any Sudoku puzzle, both manually (you solve the puzzle yourself, cell by cell) and algorithmically (you write an algorithm that solves the entire board). You will be given some methods so you can concentrate your efforts on the most important parts. In a few places, you will be given a few lines of code which are especially tricky to figure out.

You will be provided with code to initialize the board to the beginning of the puzzle. (You can write your own, if you like, and you will be given extra credit; it’s always nice when you can say, “I did this all myself.”)

**The Sudoku Board**

The Sudoku board is a static member of the class Sudoku. It is a two-dimensional array of ints.

int[,] board = new int[9, 9];

The first index is the Row in the board with values 0-8; the second index is the Column (again with values 0-8). In the boards above, cell [1, 4] contains the digit 9. The blank cells actually contain 0, indicating that no 1-9 digit has been assigned yet.

**Method: FindLegalDigits()**

The first method you need to code finds all the legal digits that can be put into a blank (zero) cell. The cell is specified by its row and column on the Sudoku board, and the method returns a list of the legal digits:

static List<int> FindLegalDigits(int row, int col)

Your code must look at all the Sudoku board cells in the specified column to find what digits are already used; none of those digits can be used in the blank cell. Similarly, the code must look at all the cells in the specified row and reject any digits found. And finally, your code must look at all 9 cells in the 3x3 box and reject any digits found.

Looking through the 3x3 box is easy once you know where the upper-left corner of the box is, so here is some help. The upper-left corner of the box is at:

int upperRowBox = row - row % 3;

int leftColBox = col - col % 3;

Your code must create and return a new list of ints, containing digits that are ***not*** used in the cell’s row, column, or box. The list might have zero entries, or nine entries, or anything in between.

This method will get executed hundreds or even millions (yes, millions) of times depending on how hard or easy the Sudoku puzzle is. If you make a coding mistake in this method, it will be incredibly hard to find. (Which of those millions went wrong?) Therefore you must test the method very thoroughly before trying to use it in more complex code. Create test cases by putting digits in the Sudoku board and then calling the FindLegalDigits() method with some test code. If the Sudoku board has all blanks (remember, blank is zero), what should FindLegalDigits() return? What if a row and column and box together have all nine digits? What if a digit is in both a row and column? (Is that legal?) What should the method return? Create your own test cases to test the method thoroughly. Do not skimp on testing: you will likely regret it later.

**Method: PrintBoard()**

You will need a method to print out or display a Sudoku board. This has been written for you but can always be improved if you want.

**Method: VerifyBoard()**

You should also write a CheckBoard() method to verify that the Sudoku rules have been followed for any solutions. Using that method is a lot easier than checking by eyeball. (And you will end up checking many times.)

public bool VerifyBoard()

The method must return true iff (read, “if and only if”) the board follows all the Sudoku rules. Test this method carefully, too.

**Method: SolveBoardIterativelyWithQueue()**

Now we come to the main algorithm to solve the Sudoku puzzle. The method looks at the Sudoku board, makes choices about what digits to put into the blank cells, detects when those choices don’t work, and then tries other digits for the blank cells. When it finally comes up with a complete, correct, and legal solution, it prints the solution and returns true.

//In Program class

public static bool SolveBoardIterativelyWithQueue(SudokuBoard board)

**Optimizations and Instrumentation**

Nobody really wants to wait a long time for a solution, so instrument your code and try some algorithm improvements that may speed things up a whole lot:

1. Instrument your code. Put a counter in FindLegalDigits(), and print that counter out when a solution is found. You will be amazed at how many times that method is called. You can also put a counter in the SolveSudoku() method.
2. Use the Stopwatch class in “using System.Diagnostics”. The Reset(), Start(), and Stop() methods and the Elapsed value will make it easy and accurate to time your code. Use Google to find out how to use these methods.
3. Try to make the code for FindLegalDigits() run as fast as possible. For difficult puzzles, this method may be executed millions of times. If you used the List.Remove() method, find another way. Remove() is slow because it must search the list to find the digit you are trying to remove.
4. In the SolveSudokuIterativelyWithQueue() method, Steps 1 and 3, find the “best” zero cell. The “best” zero cell is the one that has the fewest legal digits when your code calls FindLegalDigits(row, col). Find the minimum of the number of legal digits, and use the zero cell that has that minimum legal digit count. Save the row, column, and list of legal digits for this “best” zero cell.
5. If your code finds a zero cell with no legal digits, then return false right away. No legal solution exists with the present Sudoku board.
6. If your code finds a zero cell with exactly one legal digit, go to Step 4 right away since there are no other legal digits that can go into that cell.
7. The code can do both 5 and 6 and maybe speed things up even more. Experiment to see what gives the fastest solutions.
8. Write a method to read a file containing many Sudoku puzzles. Time how long your program takes to solve all of them total. Extra Credit.