CS 100 Programming I
Project 2

Revision Date: October 2, 2015

Solving Sudoku!

Once you have a function that checks whether or not there are any duplicates in any row, column, or section of a Sudoku grid, you are well on your way towards writing a program that can solve a partially filled-out or even an empty grid.

A Sudoku solving function is quite difficult to write using loops, but is almost laughably easy using recursion. Suppose you have a solving function named *solver*, that takes a grid, a row index, a column index, and a digit to place at that row and column. Then it becomes a simple matter: if placing the digit at the given row and column leads to a solution, you are done! If it doesn't, you need to try the next digit. If no digit works at the given location, you need try a different digit at a previous location. This business of trying a different digit at a previous location is known as *backtracking*. Recursion allows you to backtrack so easily, you don't even realizing you are doing it.

A Recurrence Equation for Solving Sudoku

Rather than giving you a formal recurrence equation for solving Sudoku (which would make this task too easy), we will describe the equation in English. Note that our solving function takes the grid, a row number, a column number, and a digit. When we wish to try a new digit at a location, we make a recursive call that changes the digit, but leaves the row and column numbers the same. When we wish to try a new column, we make a recursive call that increments the column number, keeping the row number the same, but resetting the digit back to one (since we wish to try all the digits at the new location). When we wish to work on a new row, we make a recursive call that increments the row number, but resets the column number and digit back to zero and one, respectively.

Here are the cases of the recurrence equation:

case 1: The row number has gotten too large. This means all the digits in the preceding rows have been successfully placed. This also means we are done, so we return True.

case 2: The digit has gotten too large. This means we were not able to place any digit at the current row and column successfully. This also means we have failed to find a solution, so we return False.

case 3: The column number has gotten too large. This means we were able to place digits successfully in the current row, so we need to start working on the next row. We make a recursive call that increments the row number, with column zero and digit one. We return the result of that recursive call.

case 4: The current location, as specified by the row and column number, already has a number in it. We skip this location by making a recursive call that increments the column number. Note that we will be looking at a new location, so we send one as the digit to start working with at the new location. As before, we return the result of the recursive call.

case 5: This is the backtracking step. We place the given digit at the location specified by the given row and column. Note, we know that there must have been a space where we placed the digit (why?). If (1) the newly placed digit did not cause a conflict *and* (2) we can find a solution that includes the newly placed digit, we are done and can return True. If there is a problem, however, we need to replace the space that was originally at the given location and try a new digit by making a recursive call with the next digit to try. As always, we return the result of the recursive call.

The initial call to our solver sends zero as the row number, zero as the column number, and one as the first digit to try. If our initial call returns True, we solved the puzzle. If it returns False, there is no solution.

0.1 The Tricky Bits

The only tricky bits are in case 5. The first is how to check if the newly placed digit causes a conflict. Your solution checking function can be used to tell you that. The second is how to determine if the newly placed digit, while not causing an immediate conflict, leads to a future conflict down the line. We can make a recursive call to the solver function, having it start on the next column, to tell us that. If the recursive call returns False, that means the digit we placed caused a future problem. These two checks, together, can tell us if there is a problem with the current digit at the current location; if either the current solution fails with a duplicate value or the recursive call to fill out the rest of the grid returns False, we need to backtrack by resetting the current location back to a space and trying another digit.