In this post we discuss how to write an R script to solve any Sudoku puzzle. There are some R packages to handle this, but in our case, we'll write our own solution. For our purposes, we'll assume the input Sudoku is a 9×9 grid. At the end result, each row, column, and 3×3 box needs to contain exactly one of each integer 1 through 9.

Learn more about data science by checking out the great curriculum at 365 Data Science!

Step 0) Define a sample board

Let's define a sample Sudoku board for testing. Empty cells will be represented as zeroes.

```
board <- matrix(
          c(0,0,0,0,0,6,0,0,0,
            0,9,5,7,0,0,3,0,0,
            4,0,0,0,9,2,0,0,5,
            7,6,4,0,0,0,0,0,3,
            0,0,0,0,0,0,0,0,0,
            2,0,0,0,0,0,9,7,1,
            5,0,0,2,1,0,0,0,9,
            0,0,7,0,0,5,4,8,0,
            0,0,0,8,0,0,0,0,0),

          byrow = T,
          ncol = 9
)
```

```
> board
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
[1,]    0    0    0    0    0    6    0    0    0
[2,]    0    9    5    7    0    0    3    0    0
[3,]    4    0    0    0    9    2    0    0    5
[4,]    7    6    4    0    0    0    0    0    3
[5,]    0    0    0    0    0    0    0    0    0
[6,]    2    0    0    0    0    0    9    7    1
[7,]    5    0    0    2    1    0    0    0    9
[8,]    0    0    7    0    0    5    4    8    0
[9,]    0    0    0    8    0    0    0    0    0
```

Step 1) Find the empty cells

In the first step, let's write a function that will find all of the empty cells on the board.

```
find_empty_cells <- function(board) {

  which(board == 0, arr.ind = TRUE)
```

```
}
```

Step 2) Make sure cell placement is valid

Next, we need a function that will check if a cell placement is valid. In other words, if we try putting a number into a particular cell, we need to ensure that the number appears only once in that row, column, and box. Otherwise, the placement would not be valid.

```
is_valid <- function(board, num, row, col) {

  # Check if any cell in the same row has value = num
  if(any(board[row, ] == num)) {

    return(FALSE)

  }

  # Check if any cell in the same column has value = num
  if(any(board[, col] == num)) {

    return(FALSE)

  }

  # Get cells in num's box
  box_x <- floor((row - 1) / 3) + 1
  box_y <- floor((col - 1) / 3) + 1

  # Get subset of matrix containing num's box
  box <- board[(3 * box_x - 2):(3 * box_x), (3 * box_y - 2):(3 * box_y)]

  # Check if the number appears elsewhere in its box
  if(any(box == num)) {

    return(FALSE)

  }

  return(TRUE)

}
```

Step 3) Recursively solve the Sudoku

In the third step, we write our function to solve the Sudoku. This function will return **TRUE** is the input Sudoku is solvable. Otherwise, it will return **FALSE**. The final result will be stored in a separate

variable.

```
result <- sudoku

solve_sudoku <- function(board, needed_cells = NULL, index = 1) {

  # Find all empty cells
  if(is.null(needed_cells))
      needed_cells <- find_empty(board)

  if(index > nrow(needed_cells)) {

    # Set result equal to current value of board
    # and return TRUE
    result <<- board
    return(TRUE)

  } else {

    row <- needed_cells[index, 1]
    col <- needed_cells[index, 2]
  }

  # Solve the Sudoku
  for(num in 1:9) {

    # Test for valid answers
    if(!is_valid(board, num, row, col)) {next} else{

      board2 = board
      board2[row, col] <- num

      # Retest with input
      if(solve_sudoku(board2, needed_cells, index + 1)) {
        return(TRUE)

      }

    }

  }

  # If not solvable, return FALSE
  return(FALSE)

}
```

Calling the Sudoku solver

Lastly, we call our Sudoku solver. The result is stored in the variable “result”, as can be seen below.

```
solve_sudoku(board)
```

```
> result
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
[1,]    4    3    5    2    6    9    7    8    1
[2,]    6    8    2    5    7    1    4    9    3
[3,]    1    9    7    8    3    4    5    6    2
[4,]    8    2    6    1    9    5    3    4    7
[5,]    3    7    4    6    8    2    9    1    5
[6,]    9    5    1    7    4    3    6    2    8
[7,]    5    1    9    3    2    6    8    7    4
[8,]    2    4    8    9    5    7    1    3    6
[9,]    7    6    3    4    1    8    2    5    9
```

Conclusion

That's it for this post! …