MATH 495R, HOMEWORK 6
SOLVING SUDOKU, PART 2

In this lab we will continue to modify our sudoku solving program from last week to solve *any* sudoku puzzles, not just easy ones.

We will need some preliminary functions, before writing the main routine.

1. Write a function `issolved(grid)` that checks to see if a sudoku grid is completely filled out (in other words, are all of the entries in the array nonzero). It should return TRUE if all entries have been filled in, and FALSE otherwise.
2. Write a function `issolvable(grid)` that checks to see if every entry that has not been filled out still has values that could possibly be put into it. In other words, check that for each entry in the grid that is still 0, `computepossibilities(a,b,grid)` returns a nonempty list.

We are now prepared to write a recursive function that will solve *any* solvable sudoku puzzle.

We will call the function `solve(grid)`. It will take as input a sudoku grid, and return either a completely solved sudoku grid, or the value FALSE if it fails to solve the grid.

The function should carry out the following steps.

(a) It should run `simplesolve(grid)`. After this, if the resulting grid is solved, it should return the grid. Otherwise, it should proceed to the next step.

(b) It should run `issolvable(grid)`. If the result is FALSE, the given grid cannot be solved (at least one entry cannot be filled in). The function should then return FALSE.

(c) At this point, we have a grid for which each unfilled entry has multiple possibilities (since if any entry had only one possibility, `simplesolve(grid)` would have filled it in). The function should find some entry of the grid which has not been filled out, and store the coordinates of the entry in the variables `a` and `b`. It should also compute the possible numbers that could fit into the (a, b) entry, and store than in the variable `possibilities`.

Note: If you want your program to run faster, don't just find an arbitrary entry that hasn't been filled out; instead find the one with the smallest number of possibilities. This will reduce the size of the next loop.

(d) We will now loop through the entries of `possibilities`, one at a time (use a `for` loop). For the kth entry, we will do the following steps.

  (i) Make a copy of the sudoku grid, which we will call `newgrid`.

  (ii) Use the `Set` command (from last week) to set the (a, b) entry of `newgrid` to equal the kth entry of `possibilities`.

  (iii) Check if `newgrid` is solvable. If so, run the command `newgrid=solve(newgrid)`. Otherwise do nothing.

  (iv) after running `newgrid=solve(newgrid)`, either `newgrid` will be a solved sudoku grid, or the value FALSE. If `newgrid` is not FALSE, then `return newgrid`. If `newgrid` is FALSE, there is no need to do anything; this means that the kth entry

of `possibilities` was not the correct entry; it does not lead to a solution. Continue on the the next entry of the loop.

There are two ways that we can exit this loop; we may have returned a solved sudoku (in which case, any code after the loop will never be reached), or we may have tried each possible value in the (a, b) entry, and found that none of them lead to a solution, in which case the code after the loop will be run. The code after the loop should indicate that the puzzle cannot be solved, by returning FALSE.

At this point, the function `solve(grid)` should return a solved sudoku grid for any sudoku puzzle that you put in that has a valid solution. Here are some puzzles for you to try it on. Some of them may take your program a while to solve.

003000900020904060700050003010305080006080300050209070500010008080703010009000400
980100000501000000067089000300206400004000800005708006000350260000000301000002045
010702080300050001006000700600070004090423060400080009003000900800040002050607040
903050400060008010100000007000070090300809005050060000200000006080400030006030801
400030000000600800000000001000050090080000600070200000001027005030000409000000000,
708000300000201000500000000040000026300080000001000900906000040000705000000000000
708000300000601000500000000040000026300080000001000900902000040000705000000000000
307040000000000091800000004000007000001600000002500000000003800900005000206000000
500700600003800000000002006204000000000009170000000000035080400001000000900000
400700600003800000000002006205000000000009170000000000043080500001000000900000
040010200000009070010000000004306008000005000020000705008000006003009000000000
705000002000401000300000000106004002000500000000009000037000080000600090000080
000000410900300000300050000480070000000006201000000060020000507000080000009000
705000002000401000300000000106004002000500000000009000037000090000800080000060
080010000005000300000004000006050708900002000003000020000010900670000000400000
809000300000701000500000000700000263000900000001000400602000040000805000000000
609000008000701000400000000000060004020000030030000500010500070800090000000000200
000000410900300000300020000480070000000005201000000050020000607000080000009000
100048000050000900006000300000570200803000000000900000000000041670000000000200000
708000300000601000400000000060000025300080000001000900905000020000704000000000000
Remarks:

- Unlike `simplesolve(grid)`, the function `solve(grid)` must return a value, not just change the entries of `grid`. This is because many (perhaps most) of the values that it inserts into `grid` will be guesses that are incorrect. If we actually changed the original grid with these guesses, we would have to keep track of them and be able to erase them.
- The logic of the recursive `solve(grid)` command does not actually require the function `simplesolve(grid)` at all (however, it will run faster with `simplesolve`). Try deleting the `simplesolve(grid)` command at the beginning of the function, and see what happens. (This will work better if you programmed `solve(grid)` to look for the unfilled entry with the fewest possibilities.
- Be sure that you understand the logic of the program. Could it be modified so that it could find multiple solutions to a badly formed sudoku puzzle? Try it on the following puzzle, which has two solutions.

407200000100400005000016098620300040300900000001072600002005870000600004530097061