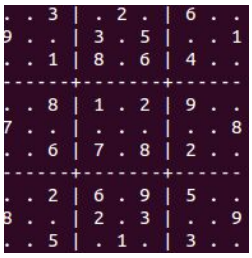
Helper functions:

Class Sudoku: builds a Sudoku problem from a string representing the grid, the digits 1-9 denote a filled cell, '.' represents an empty one; other characters are ignored.

For examples:

[easy -- can be solved by one step of AC3 in reasonable time]
'..3.2.6..9..3.5..1..18.64....81.29..7.......8..67.82....26.95..8..2.3..9..5.1.3..' (is the string version of



)
'...7.46.3..38...51.1.9.327..34...76....6.8....62...98..473.6.1.68...13..3.12.5...'

[hard -- can be solved by backtracking with AC3 in reasonable time]
'..5...1.3....2.........176.7.49....1...8.4...3....7..8.3.5....2....9....4.6...9..'

[evil -- can be solved by applying AC3 first, then backtracking with AC3 *in reasonable time*]
'4173698.5.3..........7......2.....6.....8.4......1.......6.3.7.5..2.....1.4......'

==========

CSP.prune(self, var, value, removals): removes *value* from curr_domain of *var*.
different_values_constraint(): Constraints function, returns True if two input variables are different.

These three helper functions can help you implement AC3 algorithm.
If you implement AC3 correctly, you can solve Sudoku [*easy*] examples provided above only by using the code below:

```
S = Sudoku(easy)
sol=Solver()
start=time.clock()
sol.AC3(S)
print("time: " + str(time.clock() - start))
S.display(S)
```

==========

CSP.nconflicts(self, var, val, assignment): returns the number of conflicts var=val has with other variables
CSP.assign(self, var, val, assignment): adds {var: val} to assignment
CSP.unassign(self, var, assignment): removes {var: val} from assignment
CSP.suppose(self, var, value): starts accumulating inferences from assuming var=value
CSP.restore(self, removals): undo a supposition and all inferences from it

These functions can help you implement backtracking search.
If you implement backtracking_search() function correctly, you can solve [easy] and [hard] examples:

```
S = Sudoku(hard)
sol=Solver()
start=time.clock()
sol.backtracking_search(S)
print("time: " + str(time.clock() - start))
```