Output 1:

```
============
4 3 5 2 6 9 7 8 1
6 8 2 5 7 1 4 9 3
1 9 7 8 3 4 5 6 2
8 2 6 1 9 5 3 4 7
3 7 4 6 8 2 9 1 5
9 5 1 7 4 3 6 2 8
5 1 9 3 2 6 8 7 4
2 4 8 9 5 7 1 3 6
7 6 3 4 1 8 2 5 9
```

Output 2:

```
============
1 2 3 6 7 8 9 4 5
5 8 4 2 3 9 7 6 1
9 6 7 1 4 5 3 2 8
3 7 2 4 6 1 5 8 9
6 9 1 5 8 3 2 7 4
4 5 8 7 9 2 6 1 3
8 3 6 9 2 4 1 5 7
2 1 9 8 5 7 4 3 6
7 4 5 3 1 6 8 9 2
```

Output 3:

```
============
2 7 6 3 1 4 9 5 8
8 5 4 9 6 2 7 1 3
9 1 3 8 7 5 2 6 4
4 6 8 1 2 7 3 9 5
5 9 7 4 3 8 6 2 1
1 3 2 5 9 6 4 8 7
3 2 5 7 8 9 1 4 6
6 4 1 2 5 3 8 7 9
7 8 9 6 4 1 5 3 2
```

To Run ----------

1. Have `Sudoku.py` and `SudoSolve.py` in one folder.
2. Run by using `Python3 Sudoku.py` un
3. Input file name when prompted.
4. If output file is desired, input output name. Otherwise, skip with `ENTER`.

CODE BELOW:

Sudoku.py ----------

```
import SudoSolve

# Produce entry points for search algorithm from file.
def InitFromFile(targetFile):
	initialMap = [];

	f = open(targetFile);
	fLines = f.readlines();
	f.close();

	# Getting initial map
	for line in fLines:
		nums = line.split();
		curLine = [];
		for num in nums:
			curLine.append(int(num));
		initialMap.append(curLine);

	# Generate a search instance from initial map.
	searchInstance = SudoSolve.SudokuSearch(initialMap);

	return searchInstance;

# Puts result into desired output format.
def GenerateOutput(outMap):
	out = "";

	for i in range(len(outMap)):
		for j in range(len(outMap[i])):
			out += str(outMap[i][j].value);
			out += " ";
		out += "\n";

	return out;


def main():

	# Usability
	fname = input("File Name -> ");

	# Initialize from file and create a search manager object.
	sudoSearch = InitFromFile(fname);
	result = sudoSearch.doSearch();
	output = GenerateOutput(sudoSearch.cellMap);
	output += "\n= = = = = = = = = = = = = = = = = =\n\n";
	output += GenerateOutput(result);
```

```
	print();
	print(output);

	#Write results to file if output name is specified.
	fname = "";
	fname = input("Output File Name (or ENTER to skip) -> ");
	if(fname != ""):
		writeout = open(fname, "w+");
		writeout.write(output);
		writeout.close();
		print("Saved.\n");
	else:
		print("Not saved.\n");



main();
```

SudoSolve.py ----------

```
import copy

#Variables and methods related to sudoku puzzle states.
class SudokuSearch:
	###   CLASS ATTRIBUTES   ###
	###    list cellMap      ###
	###    list iTarget      ###

	def __init__(self, inMap):
		self.cellMap = [];
		for row in inMap:
			curRow = [];
			for colVal in row:
				curCell = SudokuCell(colVal);
				curRow.append(curCell)
			self.cellMap.append(curRow);
		print(" ============================== SEARCH INSTANCE INITIATED
============================== ");

	# Returns (False) if dead end. Returns map otherwise.
	def doSearch(self, curMap=None):
		if(curMap == None):
			curMap = copy.deepcopy(self.cellMap);
		print(GenerateOutput(curMap));
		print();

		# Forwardcheck is first step in each search.
		solvable = self.ForwardCheck(curMap);
```

```
            if(solvable == False):
                  # Dead end reached.
                  print("/ / / / / / / / / / / DEAD END / / / / / / / / / / /")
                  return False;
            elif(solvable == True):
                  # Solution Found.
                  return curMap;
            else:
                  # Not dead end yet. Try to assign next target by guessing MRV
cell.
                  curTarget = [solvable[0],solvable[1]];
                  while(curMap[curTarget[0]][curTarget[1]].remVals != set()):
                        nextAttempt =
curMap[curTarget[0]][curTarget[1]].remVals.pop();

                        print("New Guessing Attempt at: ", curTarget, " with
number", nextAttempt)

                        newMap = copy.deepcopy(curMap);
                        newMap[curTarget[0]][curTarget[1]].value = nextAttempt;

                        attemptResult = self.doSearch(newMap);

                        if(attemptResult != False):
                              return attemptResult;

            return False;

      # Returns (False) if dead end. Returns (True) if solved. Returns [x,y] if
neither.
      def ForwardCheck(self, inMap):

            # Update constraints on all cells.
            if(self.UpdateConstraints(inMap)):
                  # Find next highest priority cell.
                  assigned = False;
                  target = None;
                  for i in range(9):
                        for j in range(9):
                              # If domain size is 1, simply assign value.
                              if(len(inMap[i][j].remVals) == 1 and
inMap[i][j].value == 0):
                                    target = [i,j];
                                    print("Assigning Value: ", target);
                                    inMap[target[0]][target[1]].TryAssign();
                                    assigned = True;
                              else:
                              # Domain size not one then compare domain sizes
                                    if(inMap[i][j].value == 0):
#                                         print("Checking Target at: ", [i,j])
```

```
                                                if(target == None):
                                                      target = [i,j];

elif(len(inMap[target[0]][target[1]].remVals) > len(inMap[i][j].remVals)):
                                                      print("New Target At => ",
[i,j]);
                                                      print(inMap[i][j].remVals)
                                                      target = [i,j];
                  if(assigned):
                        if(self.IsSolved(inMap)):
                              # Return (True) if the assignment completed the
puzzle.
                              return True;
                        return self.ForwardCheck(inMap);
                  else:
                        # Return the next best value to guess.
                        print("ForwardCheck returning target => ", target);
                        return target;
            else:
                  # Dead end found when updating constraints.
                  return False;

      # Returns (False) if dead end. Otherwise, returns (True).
      def UpdateConstraints(self, inMap):

            # Update Row Constraints.
            for i in range(9):
                  curConstraintSet = set();
                  for j in range(9):
                        if(inMap[i][j].value != 0):
                              curConstraintSet.add(inMap[i][j].value);
                  for j in range(9):
                        inMap[i][j].remVals = inMap[i][j].remVals -
curConstraintSet;

            # Update Column Constraints.
            for j in range(9):
                  curConstraintSet = set();
                  for i in range(9):
                        if(inMap[i][j].value != 0):
                              curConstraintSet.add(inMap[i][j].value);
                  for i in range(9):
                        inMap[i][j].remVals = inMap[i][j].remVals -
curConstraintSet;

            # Update Group Constraints
            for groupY in range (3):
                  for groupX in range (3):
                        curConstraintSet = set();
                        for i in range(3):
```

```
                                    for j in range(3):
                                          if(inMap[i+(3*groupX)][j+(3*groupY)].value !=
0):

curConstraintSet.add(inMap[i+(3*groupX)][j+(3*groupY)].value);
                              for i in range(3):
                                    for j in range(3):
                                          inMap[i+(3*groupX)][j+(3*groupY)].remVals -=
curConstraintSet;

              # Check for dead end
              for i in range(9):
                    for j in range(9):
                          if(len(inMap[i][j].remVals) == 0 and inMap[i][j].value ==
0):
                                    return False;

              return True;

       # Check if every cell has a valuea assigned.
       def IsSolved(self, inMap):
              for i in range(9):
                    for j in range(9):
                          if(inMap[i][j].value == 0):
                                return False
              return True;

# Variables and methods related to individual sudoku tiles.
class SudokuCell:
       ###   CLASS ATTRIBUTES   ###
       ###     int value        ###
       ###    list remVals      ###

       def __init__(self, iVal):
              self.value = iVal;
              if(iVal == 0):
                    self.remVals = set([1,2,3,4,5,6,7,8,9]);
              else:
                    self.remVals = set([0]);

       # Assigns value to cell with one possible value remaining.
       def TryAssign(self):
              if(len(self.remVals) == 1):
#                   print("Remaining Set: ", self.remVals);
                    self.value = self.remVals.pop();
                    print("New Value: ", self.value);
                    return True;
              print("/ ! \\ ERROR ASSIGNING!")
              return False;
```

```
# For genearting readable output format from sudoku cell map.
def GenerateOutput(outMap):
	out = "";

	for i in range(len(outMap)):
		for j in range(len(outMap[i])):
			out += str(outMap[i][j].value);
			out += " ";
		out += "\n";

	return out;
```