Before you turn this problem in, make sure everything runs as expected. First, **restart the kernel** (in the menubar, select Kernel→Restart) and then **run all cells** (in the menubar, select Cell→Run All).

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE", as well as your name and collaborators below:

In [2]:
```
NAME = ""
COLLABORATORS = ""
```

CSE 30 Spring 2022 - Homework 17

Instructions

Please disregard the YOUR NAME and COLLABORATORS above. They are put there atomatically by the grading tool. You can find instructions on how to work on a homework on Canvas. Here is a short summary:

Submitting your work

To submit your work:

- First, click on "Runtime > Restart and run all", and check that you get no errors. This enables you to catch any error you might have introduced, and not noticed, due to your running cells out of order.
- Second, download the notebook in .ipynb format (File > Download .ipynb) and upload the .ipynb file to this form. This homework is due at **11:59pm on Friday, 03 June 2022**.

You can submit multiple times; the last submission before the deadline is the one that counts.

Homework format

For each question in this notebook, there is:

- A text description of the problem.
- One or more places where you have to insert your solution. You need to complete every place marked:

```
# YOUR CODE HERE
```

and you should not modify any other place.

- One or more test cells. Each cell is worth some number of points, marked at the top. You should not modify these tests cells. The tests pass if no error is printed out: when there is a statement that says, for instance:

```
assert x == 2
```

then the test passes if `x` has value 2, and fails otherwise. You can insert a `print(x)` (for this case!) somewhere if you want to debug your work; it is up to you.

Notes:

- Your code will be tested both according to the tests you can see (the `assert` statements you can see), *and* additional tests. This prevents you from hard-coding the answer to the particular questions posed. Your code should solve the *general* intended case, not hard-code the particular answer for the values used in the tests.

- **Please do not delete or add cells!** The test is autograded, and if you modify the test by adding or deleting cells, even if you re-add cells you delete, you may not receive credit.

- **Please do not import modules that are not part of the standard library.** You do not need any, and they will likely not available in the grading environment, leading your code to fail.

- **If you are inactive too long, your notebook might get disconnected from the back-end.** Your work is never lost, but you have to re-run all the cells before you continue.

- You can write out print statements in your code, to help you test/debug it. But remember: the code is graded on the basis of what it outputs or returns, not on the basis of what it prints.

- **TAs and tutors have access to this notebook,** so if you let them know you need their help, they can look at your work and give you advice.

Grading

Each cell where there are tests is worth a certain number of points. You get the points allocated to a cell only if you pass *all* the tests in the cell.

The tests in a cell include both the tests you can see, and other, similar, tests that are used for grading only. Therefore, you cannot hard-code the solutions: you really have to solve the essence of the problem, to receive the points in a cell.

Code of Conduct

- Work on the test yourself, alone.
- You can search documentation on the web, on sites such as the Python documentation sites, Stackoverflow, and similar, and you can use the results.
- You cannot share your work with others or solicit their help.

Sudoku is a search problem. SAT is a prototypical search problem, and more precisely, SAT is perhaps the most basic of the *NP-Complete* problems: the problems where, if you could only *guess* the solution, you could *verify* that it is truly a solution in time that is polynomial in the size of the problem (in case of SAT, the sum of the length of all the clauses).

This opens the question: can we solve Sudoku, rather than by writing a special-purpose Sudoku solver, by translating the Sudoku problem to SAT, and using an off-the-shelf SAT solver? There is a lot of research that went into developing efficient SAT solvers: would it be more efficient to use a custom solver, as we did, or to translate and rely on an off-the-shelf SAT solver? Let's experiment.

Installing a SAT solver library

Let us install our library for SAT solvers. Here is a link to the pysat documentation.

In [3]:
```
try:
    import pysat
except:
    !pip install python-sat
    import pysat
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/
public/simple/
Collecting python-sat
  Downloading python_sat-0.1.7.dev17-cp37-cp37m-manylinux2010_x86_64.whl (1.8 MB)
     |████████████████████████████████| 1.8 MB 8.3 MB/s
Requirement already satisfied: six in /usr/local/lib/python3.7/dist-packages (from p
ython-sat) (1.15.0)
Installing collected packages: python-sat
Successfully installed python-sat-0.1.7.dev17
```

Let us check that our solver works. The Glucose solver is one of the solvers.

In [4]:
```
from pysat.solvers import Glucose3
g = Glucose3()
g.add_clause([-1, 2])
g.add_clause([-2, 3])
print(g.solve())
print(g.get_model())
g.delete()
```

```
True
[-1, -2, -3]
```

Rather than deleting the solver after use with `g.delete()`, it is better to use the solver within a `with` clause: this avoids the problem of forgetting to delete the solver after use.

In [5]:
```
with Glucose3() as g:
    g.add_clause([-1, 2])
    g.add_clause([-2, 3])
    print(g.solve())
    print(g.get_model())
```

```
True
[-1, -2, -3]
```

There are also Glucose4, and Minisat22. They all work in the same way.

In [6]:
```
from pysat.solvers import Glucose4, Minisat22
with Minisat22() as g:
    g.add_clause([-1, 2])
    g.add_clause([-2, 3])
    print(g.solve())
    print(g.get_model())
```

```
True
[-1, -2, -3]
```

A `SudokuViaSAT` class.

The first thing we do is to write a Sudoku class that can represent a Sudoku problem to be solved. Unlike our previosu representation, each cell here will contain either a digit 1..9, or 0, where 0 represents an unknown digit. We do not need to represent our solver's state of knowledge in terms of sets of digits, since the seach for a solution will be done in the SAT solver.

The class has three methods, which we will fill in later: one for translating the Sudoku into a SAT instance, one for solving the SAT instance, and another one for using the solution to the SAT instance to fill in the unspecified cells of the Sudoku problem.

Contrary to the previous approach, we keep the state of the board as a numpy array, of size 9 x 9; this will make indexing in the array a little bit more pleasant. The reason we could not use this representation earlier is that we wanted to associate with each cell a *set* of digits, and sets are not pleasant to represent in Numpy; single digits are.

In [7]:
```
import numpy as np

class SudokuViaSAT(object):

    def __init__(self, sudoku_string):
        """
        @param sudoku_string: an 81-long digit string: 0 represents an unknown
            digit, and 1..9 represent the respective digit.
        """
        assert len(sudoku_string) > 80
        self.board = np.zeros((9, 9), dtype=np.uint8)
        for i in range(9):
            for j in range(9):
                self.board[i, j] = int(sudoku_string[i * 9 + j])
        self.sat = None # This will be the SAT instance.
        # Perform here any other initialization you think you need.
        # YOUR CODE HERE

    def show(self):
        """Prints out the board."""
        print("+---+---+---+")
        for i in range(9):
            r = '|'
            for j in range(9):
                r += "." if self.board[i, j] == 0 else str(self.board[i, j])
                if (j + 1) % 3 == 0:
                    r += "|"
            print(r)
            if (i + 1) % 3 == 0:
                print("+---+---+---+")
```

In [8]:
```
problem = "000000061350000000040005000002000080000060100000070000000008020060040000
sd = SudokuViaSAT(problem)
sd.show()
```

```
+---+---+---+
|...|...|.61|
|35.|...|...|
|4..|.5.|...|
+---+---+---+
|.2.|...|8..|
|...|6.1|...|
|...|7..|...|
+---+---+---+
|...|.8.|2..|
|6..|4..|...|
|..7|...|.1.|
+---+---+---+
```

Variables

We base our trasnslation of Sudoku into SAT on variables p_{dij}, where p_{dij} expresses the fact that the digit d appears at coordinates (i, j). Since SAT solvers represent a variable by an integer, we will have that p_{dij} is encoded simply using the integer dij (in decimal notation), and the literal $\bar{p}_{dij}$ will be encoded as $-dij$.

For example, to express that digit 3 appears at coordinates 6, 7, we use the literal 367. To express the negation of this, $\bar{p}_{367}$, that is, that digit 3 *does not* appear at coordinates 6, 7, we use the literal -367.

We thus start by writing two helper functions, `encode_variable` and `decode_variable`, that go from d, i, j to the corresponding integer, and vice versa.
We write one of them for you, and we ask you to write the other.

In [9]:
```
def encode_variable(d, i, j):
    """This function creates the variable (the integer) representing the
    fact that digit d appears in position i, j.
    Of course, to obtain the complement variable, you can just negate
    (take the negative) of the returned integer.
    Note that it must be: 1 <= d <= 9, 0 <= i <= 8, 0 <= j <= 8."""
    assert 1 <= d <= 9
    assert 0 <= i < 9
    assert 0 <= j < 9
    # The int() below seems useless, but it is not.  If d is a numpy.uint8,
    # as an element of the board is, this int() ensures that the generated
    # literal is a normal Python integer, as the SAT solvers expect.
    return int(d * 100 + i * 10 + j)
```

In [10]:
```
# Let's define a testing helper.

def check_equal(x, y, msg=None):
    if x != y:
        if msg is None:
            print("Error:")
        else:
            print("Error in", msg, ":")
        print("    Your answer was:", x)
        print("    Correct answer: ", y)
    assert x == y, "%r and %r are different" % (x, y)
```

In [11]:
```
check_equal(encode_variable(3, 6, 7), 367)
```

It's your turn now to write a function `decode_variable` that is the opposite of `encode_variable`.

In [12]:
```
def decode_variable(p):
    """Given an integer constructed as by _create_prediate above,
    returns the tuple (d, i, j), where d is the digit, and i, j are
    the cells where the digit is.  Returns None if the integer is out of
    range.
    Note that it must be: 1 <= d <= 9, 0 <= i <= 8, 0 <= j <= 8.
    If this does not hold, return None.
    Also return None if p is not in the range from 100, to 988 (the
    highest and lowest values that p can assume).
    Hint: modulo arithmetic via %, // is useful here!"""
    # YOUR CODE HERE
    d, i, j = p//100, (p//10)%10, p%10
    try:
        assert 100 <= p <= 988
        assert 1 <= d <= 9
        assert 0 <= i < 9
        assert 0 <= j < 9
        return (d, i, j)
    except:
        return None
```

Let's test this.

In [13]:
```
for d in range(1, 10):
    for i in range(9):
        for j in range(9):
            r = decode_variable(encode_variable(d, i, j))
            check_equal((d, i, j), r)
```

Creating the clauses that represent a generic Sudoku problem

The key to translating Sudoku to SAT consists in producing a list of clauses that encodes the rules of Sudoku. We will create list of clauses expressing the following. Below, we have $1 \leq d \leq 9$, and $0 \leq i, j \leq 8$.

1. At each cell i, j at least one digit d must appear.
2. At each cell i, j, at most one digit d must appear.

- If a digit d appears at cell i, j, the same digit d will not appear elsewhere on: 3. The same column. 4. The same row. 5. The same 3x3 Sudoku block.

Note that conditions 1 and 2 are obvious to a human, and were encoded implicitly in our Sudoku solver. Our SAT solver, however, has no idea of what a variable like p_{367} means, or that digit 3 appears in cell 6, 7; therefore, we must teach it that exactly one digit apppears in each cell, via clauses.

As an example, you can say that at at least one digit appears in cell 6, 7 via the clause:

$$[p_{167}, p_{267}, \ldots, p_{967}]$$

and you can say that if 2 appears in cell 67, then 3 does not apper in that same cell, via:

$$[\bar{p}_{267}, \bar{p}_{367}] \ .$$

In literals ready for SAT, the latter is [-267, -367]. Similarly, to say that if a 2 appears at 6, 7, it does not appear on the same row at 6, 8, you would use the clause [-267, -268].

You will be creating these list of clauses below, for the cases 1, 2, 3, 4, 5 above.

1. Cells contain at least one digit

For each cell i, j, you have to create a clause stating that at least one p_{dij} is true, for some d. You can easily build it as the disjunction $p_{1ij} \vee p_{2ij} \vee \cdots \vee p_{9ij}$, corresponding to the clause:

$$[p_{1ij}, p_{2ij}, \ldots, p_{9ij}] \ .$$

Of course, to generate the clause for the SAT solver, you have to encode the variables $p_{1ij}, p_{2ij}, \ldots, p_{9ij}$ using `encode_variable`.

In [14]:
```python
def every_cell_contains_at_least_one_digit():
    """Returns a list of clauses, stating that every cell must contain
    at least one digit."""
    # YOUR CODE HERE
    for i in range(9):
        for j in range(9):
            clause = []
            for d in range(9):
                clause.append(encode_variable(d+1, i, j))
            yield clause
```

We test it with help of a SAT solver.

In [15]:
```python
def prepare(g):
    for c in every_cell_contains_at_least_one_digit():
        g.add_clause(c)

with Glucose3() as g:
    prepare(g)
    # This can be solved.
    check_equal(g.solve(), True)
    for d in range(1, 10):
        # These clauses state that no digit appears at 4, 5.
        # You can change the coordinates if you like.
        g.add_clause([-encode_variable(d, 4, 5)])
    check_equal(g.solve(), False)
```

2. Cells contain at most one digit

Next, we need to express the fact that each cell can contain at most one digit d. The idea is to write clauses that say that if a cell i, j contains a digit d, it does not contain a different digit d'. This is expressed by $p_{dij} \to \bar{p}_{d'ij}$ for all $0 \leq i, j \leq 8$ and all $1 \leq d, d' \leq 9$ with $d \neq d'$. In turn, the implication $p_{dij} \to \bar{p}_{d'ij}$ can be expressed as the clause

$$[\bar{p}_{dij}, \bar{p}_{d'ij}] \ ,$$

for all $0 \leq i, j \leq 8$ and all $1 \leq d, d' \leq 9$ with $d \neq d'$. The clause says that either d is not at i, j, or d' is not at i, j: this ensures that d, d' are not both at i, j.

In [16]:
```python
def every_cell_contains_at_most_one_digit():
    """Returns a list of clauses, stating that every cell contains
    at most one digit."""
    # YOUR CODE HERE
    for i in range(9):
        for j in range(9):
            for d in range(9):
                for notd in range(9):
                    if d != notd:
                        yield [-encode_variable(d+1, i, j), -encode_variable(notd+1
```

We test this again with the help of a SAT solver.

In [17]:
```python
def prepare(g):
    for c in every_cell_contains_at_most_one_digit():
        g.add_clause(c)



with Glucose3() as g:
    prepare(g)
    check_equal(g.solve(), True)
    # This states that both 3 and 4 appear at position 6, 7.
    g.add_clause([encode_variable(3, 6, 7)])
    g.add_clause([encode_variable(4, 6, 7)])
    check_equal(g.solve(), False)
```

3. No identical digits in the same row

We now need to experss one of the basic rules of Sudoku: a digit can appear in only one cell along a row. Precisely, for all rows $0 \leq i \leq 8$, and all digits $1 \leq d \leq 9$, we write

$$p_{dij} \rightarrow \bar{p}_{dij'}$$

for all $0 \leq j, j' \leq 8$ with $j \neq j'$. These implications stipulate that if digit d is at position j in the row, it cannot also be in position j' with $j' \neq j$. These implications can be translated into clauses with two literals, exactly as we did in point 2 above.

In [18]:
```python
def no_identical_digits_in_same_row():
    """Returns a list of clauses, stating that if a digit appears
    in a cell, the same digit cannot appear elsewhere in the
    same row, column, or 3x3 square."""
    # YOUR CODE HERE
    for i in range(9):
        for j in range(9):
            for notj in range(9):
                for d in range(9):
                    if j != notj:
                        yield [-encode_variable(d+1, i, j), -encode_variable(d+1, i
```

In [19]:
```python
def prepare(g):
    for c in no_identical_digits_in_same_row():
        g.add_clause(c)

with Glucose3() as g:
    prepare(g)
    check_equal(g.solve(), True)
    # This states that 3 appears twice in row 5.
    g.add_clause([encode_variable(3, 5, 7)])
    g.add_clause([encode_variable(3, 5, 8)])
    check_equal(g.solve(), False)

# But columns are not forbidden.
with Glucose3() as g:
    prepare(g)
    g.add_clause([encode_variable(3, 5, 7)])
    g.add_clause([encode_variable(3, 2, 7)])
    check_equal(g.solve(), True)
```

4. No identical digits in the same column

This is a similar idea to the above, but for columns.

In [20]:
```
def no_identical_digits_in_same_column():
    """Returns a list of clauses, stating that if a digit appears
    in a cell, the same digit cannot appear elsewhere in the
    same row, column, or 3x3 square."""
    # YOUR CODE HERE
    for i in range(9):
        for noti in range(9):
            for j in range(9):
                for d in range(9):
                    if i != noti:
                        yield [-encode_variable(d+1, i, j), -encode_variable(d+1, n
```

In [21]:
```
def prepare(g):
    for c in no_identical_digits_in_same_column():
        g.add_clause(c)

with Glucose3() as g:
    prepare(g)
    check_equal(g.solve(), True)
    # This states that 3 appears twice in column 7.
    g.add_clause([encode_variable(3, 5, 7)])
    g.add_clause([encode_variable(3, 2, 7)])
    check_equal(g.solve(), False)

# But rows are not forbidden.
with Glucose3() as g:
    prepare(g)
    g.add_clause([encode_variable(3, 5, 7)])
    g.add_clause([encode_variable(3, 5, 8)])
    check_equal(g.solve(), True)
```

5. No identical digits in the same 3x3 block.

The idea here is to state that if a digit d appears at a position i, j in a 3x3 Sudoku block, it does not appear in any other position i', j' in the same 3x3 block, with $i \neq i'$ or $j \neq j'$.

In [22]:
```
def no_identical_digits_in_same_block():
    """Returns a list of clauses, stating that if a digit appears
    in a cell, the same digit cannot appear elsewhere in the
    same row, column, or 3x3 square."""
    # YOUR CODE HERE
    for i in range(9):
        if i%3 == 0:
            row = [i, i+1, i+2]
        if i%3 == 1:
            row = [i-1, i, i+1]
        if i%3 == 2:
            row = [i-2, i-1, i]
        for j in range(9):
            if j%3 == 0:
                col = [j, j+1, j+2]
            if j%3 == 1:
                col = [j-1, j, j+1]
            if j%3 == 2:
                col = [j-2, j-1, j]
            for d in range(9):
                for noti in row:
                    for notj in col:
                        if i != noti or j != notj:
                            yield [-encode_variable(d+1, i, j), -encode_variable(d+
```

In [23]:

```
def prepare(g):
    for c in no_identical_digits_in_same_block():
        g.add_clause(c)

with Glucose3() as g:
    prepare(g)
    check_equal(g.solve(), True)
    # This states that 3 appears twice in top block
    g.add_clause([encode_variable(3, 1, 1)])
    g.add_clause([encode_variable(3, 1, 2)])
    check_equal(g.solve(), False)

# One more test.
with Glucose3() as g:
    prepare(g)
    g.add_clause([encode_variable(3, 1, 1)])
    g.add_clause([encode_variable(3, 2, 1)])
    check_equal(g.solve(), False)

# But different blocks are not forbidden.
with Glucose3() as g:
    prepare(g)
    g.add_clause([encode_variable(3, 1, 1)])
    g.add_clause([encode_variable(3, 5, 8)])
    check_equal(g.solve(), True)
```

Putting it all together: the rules of Sudoku.

We put this all together into a function that creates the rules for Sudoku, in SAT notation.

In [24]:

```
def sudoku_rules():
    clauses = []
    clauses.extend(every_cell_contains_at_least_one_digit())
    clauses.extend(every_cell_contains_at_most_one_digit())
    clauses.extend(no_identical_digits_in_same_row())
    clauses.extend(no_identical_digits_in_same_column())
    clauses.extend(no_identical_digits_in_same_block())
    return clauses
```

And if we solve this, we have created a Sudoku problem!

```
In [25]: with Glucose3() as g:
             for c in sudoku_rules():
                 g.add_clause(c)
             check_equal(g.solve(), True)
```

If the test above does not pass, comment out some of the lines in the `sudoku_rules` function. Try to determine which sets of clauses make the Sudoku rules unsolvable.

Translating the initial state of the board into clauses

We now need to translate the intial state of the board into clauses. This is easy to do: whenever the board contains a (known) digit d in position i, j, you generate a clause

$$[p_{dij}]$$

stating that d is in position i, j. That's all! We let you do implement this via a method `_board_to_SAT` of `SudokuViaSAT`, which returns the list of such unary clauses.

```
In [26]: def _board_to_SAT(self):
             """Translates the currently known state of the board into a list of SAT
             clauses.  Each clause has only one literal, and expresses the fact that a
             given digit is in a given position.  The method returns the list of clauses
             corresponding to all the initially known Sudoku digits."""
             # YOUR CODE HERE
             clauses = []
             for i in range(9):
                 for j in range(9):
                     if self.board[i, j] != 0:
                         clauses.append([encode_variable(self.board[i, j], i, j)])
             return clauses


         SudokuViaSAT._board_to_SAT = _board_to_SAT
```

Let us test this.

```
In [27]: problem = "000000061350000000040005000002000080000006010000007000000000080200600400000
         sd = SudokuViaSAT(problem)
         sd.show()
         clauses = sd._board_to_SAT()
         check_equal(len(clauses), 17)
         check_equal([310] in clauses, True)
         check_equal([511] in clauses, True)
         check_equal([224] in clauses, False)

         # This should print the elements of the board.
         for c in sd._board_to_SAT():
             print(c)
```

```
+---+---+---+
|...|...|.61|
|35.|...|...|
|4..|.5.|...|
+---+---+---+
|.2.|...|8..|
|...|6.1|...|
|...|7..|...|
+---+---+---+
|...|.8.|2..|
|6..|4..|...|
|..7|...|.1.|
+---+---+---+
[607]
[108]
[310]
[511]
[420]
[524]
[231]
[836]
[643]
[145]
[753]
[864]
[266]
[670]
[473]
[782]
[187]
```

Translating Sudoku to SAT

We now write a `_to_SAT` method for `SudokuViaSAT`, that translates a Sudoku problem into a list of SAT clauses, and returns the list of clauses. The list contains:

- all the clauses returned by the `sudoku_rules` function above,
- all the clauses that represent the initial state of the board, returned by the `_board_to_SAT` method.

In [28]:
```python
def _to_SAT(self):
    return list(sudoku_rules()) + list(self._board_to_SAT())

SudokuViaSAT._to_SAT = _to_SAT
```

Let's try if we can solve an instance of Sudoku via SAT.

In [29]:
```python
problem = "000000061350000000400050000020000800000601000000700000000080200600400000
sd = SudokuViaSAT(problem)
with Glucose3() as g:
    for c in sd._to_SAT():
        g.add_clause(c)
    check_equal(g.solve(), True)
```

```
problem = "000000613500000000404050000020000800000601000000700000000080200600400000
sd = SudokuViaSAT(problem)
with Glucose3() as g:
    for c in sd._to_SAT():
        for j in c:
            g.add_clause(c)
    check_equal(g.solve(), False)
```

Indeed it works!

Writing a `solve` method for Sudoku

It is time to put everything together in a `solve` method for `SudokuViaSAT`. The method works as follows. It takes as input one of the SAT solver classes, such as `Glucose3`, `Glucose4`, or `Minisat22`. Then:

- It uses the method `_to_SAT` to create the clauses for a SAT instance encoding the Sudoku problem.
- It adds those clauses to the SAT solver.
- It solves the SAT problem.
- If the problem has a solution, it uses the solution of the SAT problem to complete the cell in the Sudoku board.

For the last step, we can check that the problem has a solution via `g.solve()`, as in the test cases above. If the problem has a solution, `g.get_model()` gives us a truth assignment satisfying the SAT problem. Let us take a look at it.

In [30]:
```
problem = "000000613500000000400050000020000800000601000000700000000080200600400000
sd = SudokuViaSAT(problem)
with Glucose3() as g:
    for c in sd._to_SAT():
        g.add_clause(c)
    check_equal(g.solve(), True)
    # Let's get a truth assignment.
    ps = g.get_model()
    print(ps)
```

```
[-1, -2, -3, -4, -5, -6, -7, -8, -9, -10, -11, -12, -13, -14, -15, -16, -17, -18, -1
9, -20, -21, -22, -23, -24, -25, -26, -27, -28, -29, -30, -31, -32, -33, -34, -35, -
36, -37, -38, -39, -40, -41, -42, -43, -44, -45, -46, -47, -48, -49, -50, -51, -52,
-53, -54, -55, -56, -57, -58, -59, -60, -61, -62, -63, -64, -65, -66, -67, -68, -69,
-70, -71, -72, -73, -74, -75, -76, -77, -78, -79, -80, -81, -82, -83, -84, -85, -86,
-87, -88, -89, -90, -91, -92, -93, -94, -95, -96, -97, -98, -99, -100, -101, -102, -
103, -104, -105, -106, -107, 108, -109, -110, -111, 112, -113, -114, -115, -116, -11
7, -118, -119, -120, -121, -122, 123, -124, -125, -126, -127, -128, -129, 130, -131,
-132, -133, -134, -135, -136, -137, -138, -139, -140, -141, -142, -143, -144, 145, -
146, -147, -148, -149, -150, -151, -152, -153, -154, -155, 156, -157, -158, -159, -1
60, 161, -162, -163, -164, -165, -166, -167, -168, -169, -170, -171, -172, -173, 17
4, -175, -176, -177, -178, -179, -180, -181, -182, -183, -184, -185, -186, 187, -18
8, -189, -190, -191, -192, -193, -194, -195, -196, -197, -198, -199, 200, -201, -20
2, -203, -204, -205, -206, -207, -208, -209, -210, -211, -212, -213, -214, -215, -21
6, 217, -218, -219, -220, -221, -222, -223, -224, 225, -226, -227, -228, -229, -230,
231, -232, -233, -234, -235, -236, -237, -238, -239, -240, -241, -242, -243, 244, -2
45, -246, -247, -248, -249, -250, -251, -252, -253, -254, -255, -256, -257, 258, -25
9, -260, -261, -262, -263, -264, -265, 266, -267, -268, -269, -270, -271, 272, -273,
-274, -275, -276, -277, -278, -279, -280, -281, -282, 283, -284, -285, -286, -287, -
288, -289, -290, -291, -292, -293, -294, -295, -296, -297, -298, -299, -300, -301, -
302, 303, -304, -305, -306, -307, -308, -309, 310, -311, -312, -313, -314, -315, -31
6, -317, -318, -319, -320, -321, -322, -323, -324, -325, -326, -327, 328, -329, -33
0, -331, -332, -333, -334, 335, -336, -337, -338, -339, -340, 341, -342, -343, -344,
-345, -346, -347, -348, -349, -350, -351, -352, -353, -354, -355, -356, 357, -358, -
359, -360, -361, 362, -363, -364, -365, -366, -367, -368, -369, -370, -371, -372, -3
73, -374, -375, 376, -377, -378, -379, -380, -381, -382, -383, 384, -385, -386, -38
7, -388, -389, -390, -391, -392, -393, -394, -395, -396, -397, -398, -399, -400, -40
1, -402, -403, -404, 405, -406, -407, -408, -409, -410, -411, -412, -413, -414, -41
5, -416, -417, 418, -419, 420, -421, -422, -423, -424, -425, -426, -427, -428, -429,
-430, -431, 432, -433, -434, -435, -436, -437, -438, -439, -440, -441, -442, -443, -
444, -445, 446, -447, -448, -449, -450, -451, -452, -453, 454, -455, -456, -457, -45
8, -459, -460, -461, -462, -463, -464, -465, -466, 467, -468, -469, -470, -471, -47
2, 473, -474, -475, -476, -477, -478, -479, -480, 481, -482, -483, -484, -485, -486,
-487, -488, -489, -490, -491, -492, -493, -494, -495, -496, -497, -498, -499, -500,
-501, -502, -503, -504, -505, 506, -507, -508, -509, -510, 511, -512, -513, -514, -5
15, -516, -517, -518, -519, -520, -521, -522, -523, 524, -525, -526, -527, -528, -52
9, -530, -531, -532, 533, -534, -535, -536, -537, -538, -539, -540, -541, -542, -54
3, -544, -545, -546, -547, 548, -549, -550, -551, 552, -553, -554, -555, -556, -557,
-558, -559, 560, -561, -562, -563, -564, -565, -566, -567, -568, -569, -570, -571, -
572, -573, -574, -575, -576, 577, -578, -579, -580, -581, -582, -583, -584, 585, -58
6, -587, -588, -589, -590, -591, -592, -593, -594, -595, -596, -597, -598, -599, -60
0, -601, -602, -603, -604, -605, -606, 607, -608, -609, -610, -611, -612, -613, 614,
-615, -616, -617, -618, -619, -620, -621, 622, -623, -624, -625, -626, -627, -628, -
629, -630, -631, -632, -633, -634, -635, -636, -637, 638, -639, -640, -641, -642, 64
3, -644, -645, -646, -647, -648, -649, -650, 651, -652, -653, -654, -655, -656, -65
7, -658, -659, -660, -661, -662, -663, -664, 665, -666, -667, -668, -669, 670, -671,
-672, -673, -674, -675, -676, -677, -678, -679, -680, -681, -682, -683, -684, -685,
686, -687, -688, -689, -690, -691, -692, -693, -694, -695, -696, -697, -698, -699, -
700, -701, -702, -703, 704, -705, -706, -707, -708, -709, -710, -711, -712, -713, -7
14, -715, 716, -717, -718, -719, -720, 721, -722, -723, -724, -725, -726, -727, -72
8, -729, -730, -731, -732, -733, -734, -735, -736, 737, -738, -739, 740, -741, -742,
-743, -744, -745, -746, -747, -748, -749, -750, -751, -752, 753, -754, -755, -756, -
757, -758, -759, -760, -761, -762, -763, -764, -765, -766, -767, 768, -769, -770, -7
71, -772, -773, -774, 775, -776, -777, -778, -779, -780, -781, 782, -783, -784, -78
5, -786, -787, -788, -789, -790, -791, -792, -793, -794, -795, -796, -797, -798, -79
9, -800, 801, -802, -803, -804, -805, -806, -807, -808, -809, -810, -811, -812, 813,
-814, -815, -816, -817, -818, -819, -820, -821, -822, -823, -824, -825, -826, 827, -
828, -829, -830, -831, -832, -833, -834, -835, 836, -837, -838, -839, -840, -841, 84
2, -843, -844, -845, -846, -847, -848, -849, -850, -851, -852, -853, -854, 855, -85
6, -857, -858, -859, -860, -861, -862, -863, 864, -865, -866, -867, -868, -869, -87
0, -871, -872, -873, -874, -875, -876, -877, 878, -879, 880, -881, -882, -883, -884,
-885, -886, -887, -888, -889, -890, -891, -892, -893, -894, -895, -896, -897, -898,
-899, -900, -901, 902, -903, -904, -905, -906, -907, -908, -909, -910, -911, -912, -
913, -914, 915, -916, -917, -918, -919, -920, -921, -922, -923, -924, -925, 926, -92
7, -928, -929, -930, -931, -932, -933, 934, -935, -936, -937, -938, -939, -940, -94
1, -942, -943, -944, -945, -946, 947, -948, -949, 950, -951, -952, -953, -954, -955,
-956, -957, -958, -959, -960, -961, -962, 963, -964, -965, -966, -967, -968, -969, -
970, 971, -972, -973, -974, -975, -976, -977, -978, -979, -980, -981, -982, -983, -9
84, -985, -986, -987, 988]
```

This truth assignment contains:

- Garbage. For some odd reason, the SAT solvers want to give us truth assignments also to variables that are not part of any clause, such as 1, 2, 3, ...
- Negative literals, such as -456. We really don't care to know that 4 cannot appear in cell 5,6.
- Positive literals that can be interpreted via `decode_variable` (defined at the beginning). These we use to completethe board. For instance, if we get a literal 345 in the model, this means that 3 appears in cell 4, 5, and we can set `self.board[4, 5] = 3`.

Let's take a look at the positive, interpretable literals:

```
In [31]: for l in ps:
             if l > 0 and decode_variable(l) is not None:
                 print(decode_variable(l))
```

```
(1, 0, 8)
(1, 1, 2)
(1, 2, 3)
(1, 3, 0)
(1, 4, 5)
(1, 5, 6)
(1, 6, 1)
(1, 7, 4)
(1, 8, 7)
(2, 0, 0)
(2, 1, 7)
(2, 2, 5)
(2, 3, 1)
(2, 4, 4)
(2, 5, 8)
(2, 6, 6)
(2, 7, 2)
(2, 8, 3)
(3, 0, 3)
(3, 1, 0)
(3, 2, 8)
(3, 3, 5)
(3, 4, 1)
(3, 5, 7)
(3, 6, 2)
(3, 7, 6)
(3, 8, 4)
(4, 0, 5)
(4, 1, 8)
(4, 2, 0)
(4, 3, 2)
(4, 4, 6)
(4, 5, 4)
(4, 6, 7)
(4, 7, 3)
(4, 8, 1)
(5, 0, 6)
(5, 1, 1)
(5, 2, 4)
(5, 3, 3)
(5, 4, 8)
(5, 5, 2)
(5, 6, 0)
(5, 7, 7)
(5, 8, 5)
(6, 0, 7)
(6, 1, 4)
(6, 2, 2)
(6, 3, 8)
(6, 4, 3)
(6, 5, 1)
(6, 6, 5)
(6, 7, 0)
(6, 8, 6)
(7, 0, 4)
(7, 1, 6)
```

```
(7, 2, 1)
(7, 3, 7)
(7, 4, 0)
(7, 5, 3)
(7, 6, 8)
(7, 7, 5)
(7, 8, 2)
(8, 0, 1)
(8, 1, 3)
(8, 2, 7)
(8, 3, 6)
(8, 4, 2)
(8, 5, 5)
(8, 6, 4)
(8, 7, 8)
(8, 8, 0)
(9, 0, 2)
(9, 1, 5)
(9, 2, 6)
(9, 3, 4)
(9, 4, 7)
(9, 5, 0)
(9, 6, 3)
(9, 7, 1)
(9, 8, 8)
```

Aha! This looks very good! as you can see from the end, there are 9 tuples beginning with 9; these are exactly the location of the digits "9" in the Sudoku. Thus, the model of the SAT solver enables us to directly fill the Sudoku board. We let you implement the `solve` method.

In [32]:
```python
def solve(self, Solver):
    """Solves the Sudoku instance using the given SAT solver
    (e.g., Glucose3, Minisat22).
    @param Solver: a solver, such as Glucose3, Minisat22.
    @returns: False, if the Sudoku problem is not solvable, and True, if it is.
        In the latter case, the solve method also completes self.board,
        using the solution of SAT to complete the board."""
    # YOUR CODE HERE
    with Solver() as s:
        for c in self._to_SAT():
            s.add_clause(c)
        if s.solve() == True:
            for l in s.get_model():
                if l > 0 and decode_variable(l) is not None:
                    d, i, j = decode_variable(l)
                    self.board[i, j] = d
            return True
        else:
            return False

SudokuViaSAT.solve = solve
```

Let's try solving a Sudoku end-to-end.

In [33]:
```
problem = "000000061350000000400050000020000800000601000000700000000080200600400000
sd = SudokuViaSAT(problem)
sd.show()
sd.solve(Glucose3)
sd.show()
```

```
+---+---+---+
|...|...|.61|
|35.|...|...|
|4..|.5.|...|
+---+---+---+
|.2.|...|8..|
|...|6.1|...|
|...|7..|...|
+---+---+---+
|...|.8.|2..|
|6..|4..|...|
|..7|...|.1.|
+---+---+---+
+---+---+---+
|289|374|561|
|351|869|724|
|476|152|983|
+---+---+---+
|124|593|876|
|738|621|495|
|965|748|132|
+---+---+---+
|513|986|247|
|692|417|358|
|847|235|619|
+---+---+---+
```

This is quite wonderful! Let us write a check that a solved Sudoku satisfies all rules of Sudoku, and is equal to a given problem in the specified cells.

In [34]:
```
def verify_solution(Solver, sudoku_string):
    sd = SudokuViaSAT(sudoku_string)
    sd.solve(Solver)
    # Check that we leave alone the original assignment.
    for i in range(9):
        for j in range(9):
            d = int(sudoku_string[i * 9 + j])
            if d > 0:
                assert sd.board[i, j] == d
    # Check that there is a digit in every cell.
    for i in range(9):
        for j in range(9):
            assert 1 <= sd.board[i, j] <= 9
    # Check the exclusion rules of Sudoku.
    for i in range(9):
        for j in range(9):
            # No repetition in row.
            for ii in range(i + 1, 9):
                assert sd.board[i, j] != sd.board[ii, j]
            # No repetition in column.
            for jj in range(j + 1, 9):
                assert sd.board[i, j] != sd.board[i, jj]
            # No repetition in block
            ci, cj = i // 3, j // 3
            for bi in range(2):
                ii = ci * 3 + bi
                for bj in range(2):
                    jj = cj * 3 + bj
                    if i != ii or j != jj:
                        assert sd.board[i, j] != sd.board[ii, jj]
```

In [35]:

```
verify_solution(Glucose3, problem)
```

Let's check that you get the first 100 puzzles right.

In [36]:

```
import requests

r = requests.get("https://raw.githubusercontent.com/shadaj/sudoku/master/sudoku17.t
puzzles = r.text.split()
```

In [ ]:

```
for problem in puzzles[:100]:
    verify_solution(Glucose3, problem)
```

What is faster: custom solution, or solving via SAT?

Let us compute the time it takes us to solve a problem with each solver.

In [ ]:

```
import time

def compute_performance(problem_idx):
    for Solver in [Glucose3, Glucose4, Minisat22]:
        t = time.time()
        sd = SudokuViaSAT(puzzles[problem_idx])
        sd.solve(Solver)
        dt = time.time() - t
        print(Solver.__name__, ":", dt)
```

Puzzle 6361 was taking 0.013 with our custom solver.

In [ ]:

```
compute_performance(6361)
```

Our custom solver does a little better. Let's try other hard problems. Using our custom solver, we had:

```
Idx: 1258 d: 0.7234883308410645
Idx: 5302 d: 1.112680435180664
Idx: 10980 d: 1.6263811588287354
Idx: 25632 d: 1.7625515460968018
Idx: 48287 d: 2.2216758728027344
```

Let us see how we do on these.

```
In [ ]: compute_performance(1258)
        compute_performance(5302)
        compute_performance(10980)
        compute_performance(25632)
        compute_performance(48287)
```

Ha! This is quite interesting. What our solver found hard, these SAT solvers find easy! Let's see the worst case of Glucose4 on all the testset:

```
In [ ]: def compute_performance(Solver, num_problems):
            max_time = 0
            for idx, problem in enumerate(puzzles[:num_problems]):
                t = time.time()
                sd = SudokuViaSAT(problem)
                sd.solve(Solver)
                dt = time.time() - t
                if dt > max_time:
                    print(idx, ":", dt)
                    max_time = dt
```

Let's try the first 1000:

```
In [ ]: compute_performance(Glucose4, 1000)
```

```
0 : 0.05917024612426758
2 : 0.11767578125
6 : 0.11866617202758789
50 : 0.12514877319335938
67 : 0.17494797706604004
71 : 0.1821577548980713
75 : 0.19161677360534668
```

If you experimented for a while, you would notice that solving Sudoku via SAT has a worse average-case behavior, but better worst-case behavior, than our custom solver. This is not bad at all for an approach that relies on an off-the-shelf rather than a custom solver. This helps illustrate the practical importance of efficient SAT solvers as tools that can be used to solve a wide range of search problems.