Exercise B.1

(a) Model the Sudoku problem as a propositional satisfiability problem. All constraints need to be expressed in conjunctive normal form (CNF). Your formalization needs to be general, that is, work for any possible initial configuration of the Sudoku board.

To model the Sudoku problem into CNF $9x9x9 = 729$ propositional variables are required. That is, for every entry in the $9x9$ grid G, we associate 9 variables. We decided to use the notations G_{rcn} to refer to each needed variable. Variable G_{rcn} is assigned true if and only if the entry in row r and column c is assigned number n. Hence, $G_{483} = 1$ means that $G[4; 8] = 3$. Needless to say, the preassigned entries of the Sudoku grid will be represented as unit clauses. The constraints that are going to be added to the SAT encoding are for each entry, each row, each column and each subgrid of size $3x3$. We came up with the following minimal encoding, asserting that there is at least one number in each entry, and that each number appears at most once in each row, in each column and in each subgrid.

- There is at least one number in each entry:
  $\bigwedge_{r=1}^{9} \bigwedge_{c=1}^{9} \bigwedge_{n=1}^{9} S_{rcn}$
- Each number appears at most once in each row:
  $\bigwedge_{c=1}^{9} \bigwedge_{n=1}^{9} \bigwedge_{r=1}^{8} \bigwedge_{i=r+1}^{9} (\neg S_{rcn} \vee \neg S_{icn})$
- Each number appears at most once in each column:
  $\bigwedge_{r=1}^{9} \bigwedge_{n=1}^{9} \bigwedge_{c=1}^{8} \bigwedge_{i=c+1}^{9} (\neg S_{rcn} \vee \neg S_{rin})$
- Each number appears at most once in each subgrid:
  $\bigwedge_{n=1}^{9} \bigwedge_{i=0}^{2} \bigwedge_{j=0}^{2} \bigwedge_{r=1}^{3} \bigwedge_{c=1}^{3} \bigwedge_{k=c+1}^{3} (\neg S_{(3i+r)(3j+c)n} \vee \neg S_{(3i+r)(3j+k)n})$

  $\bigwedge_{n=1}^{9} \bigwedge_{i=0}^{2} \bigwedge_{j=0}^{2} \bigwedge_{r=1}^{3} \bigwedge_{c=1}^{3} \bigwedge_{k=r+1}^{3} \bigwedge_{l=1}^{3} (\neg S_{(3i+r)(3j+c)n} \vee \neg S_{(3i+k)(3j+l)n})$

The resulting CNF formula will have 8829 clauses,apart from the unit clauses representing the preassigned entries. Of these clauses, 81 clauses have 9 possible values and the remaining 8748 clauses are binary. The ones with 9 possible values clauses result from the set of at-least-one clauses ($9x9 = 81$),whereas the 8748 binary clauses result from the three sets of at-most-one clauses($3x9x9xC_{9,2}$).

(b) Write a Sudoku solver that makes use of the above compilation to propositional satisfiability.

For the coding solution, see attached code and README.md file.

(c) Write another program that counts the number of possible solutions (i.e., completions) of a given Sudoku board. Play a bit with your new "solution counter" and some of the instances you can find in the benchmarks directory. How many solutions do these instances usually have? What happens if you remove some of the pre-filled digits? How many of them can you remove and still have your solver report the number of solutions under a reasonable amount of time, e.g. one minute?

For the coding solution, see attached code and README.md file.

After trying several of the configurations provided inside the `benchmarks` folder we realise that most of them (if not all) take more than one minute to found all solutions. As per the sake of an easy testing, we decided to include a threshold taking into account a reasonable limit as the one suggested (1 minute) to observe faster whether this limit was reached and if so, if all solutions were found or not. We provide below 2 examples to show our findings:

- Example: .....1.....2.3.....3.2...1.4.....5....6..3.....4.7..8...962...7...

  Found **1** possible solutions below the search time **limit of one minute**, taking 0.01 seconds in total. After removing some of the values (final string was .....1.....2.3.....3.2...1.4.....5....6..3.....8...962...7...) found 1020 solutions within the restriction of one minute. After increasing the restriction up to **10** minutes, the number of solutions found was **2906**. We also wanted to check using C-level implementation of the `pycosat` library to contrast the results and its fastness. The process needed to be interrupted as it was running already for **222 minutes and 57 seconds**.

- Example: 98.76.5..7..4..8....6..3...8...79.54...5....8.....9..2..9..4...1..2.....7.2.

  Found **1** possible solutions below the search time **limit of one minute**. After removing some of the values (final string was 98.76.5..7..4..8....6..3...8...79.54...5....8.....4...1..2.....7.2.) found 989 solutions within the restriction of one minute. After increasing the restriction up to **10** minutes, the number of solutions found was **2744**. The same as above, we wanted to check using C-level implementation of the `pycosat` library to contrast the results and its fastness. In this case, the process was able to found **57562** solutions in **289.71** seconds.

Intuitively, one can imagine that by removing some of the pre-filled digits more solutions can be found as you increase the number of possible variable combinations (that is, more different clauses) by reducing the number of restrictions (that is, the initial Sudoku grid configuration).
As our implementation of obtaining all possible solutions is a naive one (that is, using the simplest solution by not knowing that a better one could exist in terms of time) we recognize that the huge amount of time taking into find them is related to this naivety. Firstly, it is quite slow as `pycosat.solve` has to convert the list of clauses over and over and over again. Secondly, after calling it recursively the list of clauses will be modified. However, in our case we use a SAT Solver solution provided by `pycosat` Python library that also offers a faster implementation done in C level, taking advantage and use of the `pycosat` C interface, see below:

```
len(list(pycosat.itersolve(cnf)))
>> 18
```

Nevertheless, we wanted to understand better the complexity in time of the proposed loop approach by performing a naively implementation of finding all solutions.

(d) What is the size of your SAT problem (number of variables, number of CNF clauses)? How does that size depend on the size n of the Sudoku board side (assuming hypothetically that your program could work for arbitrary sizes of the Sudoku board!) Run your solver against a large number of instances from any of the two provided benchmarks. What are the runtimes you observe? What conclusions can you draw from them?

As it was stated previously, encoding the Sudoku puzzle into the CNF required 729 propositional variables. Each on the entries on the grid is associated with 9 variables. The constraints used were the digit of the cell (each cell), each row and each column. In this case, the size of the encoding is $O(n^4)$, where n refers to the order of the Sudoku puzzle. This was the size of our minimal encoding, but even for an extended encoding the added constraints would have been redundant and so for both encodings the size is $O(n^4)$. The resulting CNF formula has 8829 clauses, apart from the pre-assigned entries. An extended encoding would have 11988 clauses. The Sudoku puzzle solver takes 0.015625 seconds to run. Compared to the benchmarks it is interesting to see that as the size increases so does the running time. This is because puzzles with a larger size include more variables and clauses and so run time will increase. For example, if we take a puzzle of size $36x36$, then the number of variables would be 46656 and so the running time would be higher than our simple $9x9$ Sudoku puzzle mentioned earlier on. As a result, the size of the puzzle is directly proportional to the run-time, if the size is larger, then the number of variables and clauses are higher and so the run-time will be higher as well.