Question-1:

Implementation Details:

Propositions are defined using 4 variables row, column, value and sudoku_index.
For ex: v[0][0][3], sudoku_index=0 means is true iff the cell corresponding to the first row and first column of the first sudoku has value 4(3+1).
v^v v ∨ v
We encode the problem into a sat solver as follows:

- Each cell is assigned a value. i.e.
  $\forall i, j(v[i][j][0] \vee v[i][j][1] \vee v[i][j][2] \ldots\ldots \vee v[i][j][(k * k) - 1])$
- Each cell is assigned at most one value. i.e.
  $\forall i, j, val1, val2 \sim (v[i][j][val1] \wedge v[i][j][val2])\ where\ val1 \neq val2$
- One value is assigned to at least one cell in a row. i.e.
  $\forall val, j(v[0][j][val] \vee v[1][j][val] \vee v[2][j][val] \ldots\ldots \vee v[(k * k) - 1][j][val])$
- Two cells in a row do not have the same value. i.e.
  $\forall i, val, j1, j2 \sim (v[i][j1][val] \wedge v[i][j2][val])\ where\ val1 \neq val2$
- Similarly for columns and required (k*k) boxes.
- Corresponding cells of 2 sudokus do not have the same value.
  $\forall i, j \sim ((v[i][j], M = 0) \wedge (v[i][j], M = 1))$
  The ~(a $\wedge$ b) statements can written with an or as(~a $\vee$~b).

Now, we assign a unique_id to each proposition using the *cnum* function(discussed later) in order to create clauses in pysat.
This is achieved using the following functions created:

- **cnum(a,b,c)**: Takes 3 numbers a,b,c corresponding to the row_index, column_index and (value of the cell-1), another variable M(0 or 1) is used to index the 2 sudokus of a pair. These then return a unique_id.
  unique_id=2*(a*k^4+b*k^2+c+1)+M
  This unique_id defines the literal corresponding to the following statement:
  "The cell[a][b] of sudoku M has value c+1."
- **inv(i)**: An inverse function which takes the unique_id and returns the tuple(a,b,c).
- **get_sol(assump)**: Takes a list of assumptions for the pysat Solver and gives two (k*k) sudokus pair.
- **unq(x)**: Takes a list of literals as input and then adds clauses of the form [-i,-j] to the solver such that (i>j), where i,j belong to x, i.e. (i,j) are not true simultaneously. Thus, ensuring that utmost one of the literals in x can be true at a time.
  This is used as a helper function for make_sudoku().
- **make_sudoku()**: This function adds the following types of clauses to the solver:
  - Every cell of a sudoku has at least one value lying in the interval [1,k*k] and unq(x) is used to ensure that each cell has at most one value in the interval [1,k*k].
    This is done as follows for cell[x][y]:
    *for num in [0,k*k):*

```
        x.append(cnum(x,y,num))
    solver.add_clause(x)
    unq(x)
```

- Every element of a row is unique and at least one element of the row has value num. This is done as follows:

```
for x in [0,k*k):
    x.append(cnum(x,y,num))
solver.add_clause(x)
unq(x)
```

  Other conditions can be obtained by iterating over y and num.
- Similarly for every column and for the required (k*k) blocks.

In order to solve the sudoku we do the following:

- Set M=0. Call make_sudoku()
- Set M=1. Call make_sudoku()
- Add Clauses to the solver such that corresponding entries of the 2 sudokus are not the same for any cell.
- Append literals to the assump list based on the read csv containing the unsolved sudoku
- Call get_sol(assump).

Limitations:

Solution complexity increases very strongly with k. Hence, making it infeasible to run for higher values of k (k>7).

Question-2:

Implementation Details:

1. The functions used are the same as Question-1.
2. In order to introduce randomness, we generate a random permutation of numbers from 1 to (k*k).
3. This permutation is assigned to the first row of the first sudoku and the sudoku pair is solved under these assumptions by using question 1.
4. Now, we assign the literals corresponding to our solution to the assump (assumptions) list.
5. Maintain a list assump_un, such that for all i,j if value[i][j][M0] =c , then add -cnum(i,j,c-1) at M=M0 to the list.
6. Append assump_un as a clause to the solver in addition to the clauses added during solving stage of step 3.
7. Now solver returns true if their exists a solution such that not all values are same corresponding to the solution in step 3.
8. We begin to iterate over the assump list as follows:
   For the current element, I check if the sudoku can be uniquely solved if I remove this literal, if so then I remove the literal making it a hole, otherwise I add this literal to the clauses of the solver.

9. We claim that this algorithm generates a maximal sudoku pair.
    - Proof: Let us assume that there exists some literal in the assump list which we did not remove during the iteration and it is now it is possible to remove that literal whilst maintaining the uniqueness of the sudoku pair solution.
      If so, then we must have left that literal untouched in our first iteration over the assump list. This implies that removing that literal during our first iteration would have resulted in more than one solution of the sudoku pair.
      Now, all the solutions obtained at that stage would also satisfy our sudoku pair, hence resulting in more than one solution. Contradiction.
      Therefore, the sudoku pair generated by our algorithm must be maximal.

Limitations:
Solution complexity increases very strongly with k. Hence, making it infeasible to run for higher values of k ($k>6$).