Implementation Details:

Part 1:

- We map the Boolean variable x[i][j][v] which stores if the value at ith row, jth column of the first sudoku is v or not to a unique integer (((i)*(k**4)) + ((j)*k*k) + v + 1). Similarly, for 2nd sudoku we have y[i][j][v] mapped to (((i)*(k**4)) + ((j)*k*k) + v + k**6 + 1)
- We take in a csv file containing both the partially filled sudoku, delimited by commas (‘,’), parse it and store it as a list.
- We construct the list of clauses for both the sudokus for the following conditions:
    - A cell must have a non-zero value.
    - A cell must not have more than one nonzero value.
    - Two cells of the same column can’t have the same value.
    - Two cells of the same row can’t have the same value.
    - Two cells of the same sub-grid(k*k) can’t have the same value.
    - Both x[i][j][v] and y[i][j][v] can’t be one at the same time. (Pair constraint)
    - The variables corresponding to the entered sudoku must be 1.
- We then invoke a solver object of pySAT and use the solve() method.
- If the return value is None then we print that a sudoku pair is not possible for the given pair of inputs and exit. Else we print (one of the possible) solution as returned by the solver at the console and also export it to a csv file named “Output.csv”.

Part 2:

- Similar to part 1, we map the variables x[i][j][v] and y[i][j][v] to integers and add all the clauses except the last from the previous list.
- We then need to generate a random, solved sudoku pair which we do as follows:
    - We generate a list (v[]) from (1 to k*k) and shuffle it with the random.shuffle() method. Similarly, we make another list (rand[]) from (0 to k-1) and shuffle it.
    - With help of the list rand[] we select a unique column range (rand[i0]*k,(rand[i0]+1)*k) for a particular row range (i0*k,(i0+1)*k) where i0 runs from 0 to k-1. This gives us a sub-grid (k*k) of the sudoku with a unique column and row number.
    - We then assign values in random order to the cells of this sub-grid using the list v[]. After assigning, we shuffle the values of v[] before iterating to the next.
    - We take the sudoku obtained from above steps as our first sudoku. We take the second sudoku as a blank sudoku (to assure a high probability that a sudoku pair exists). Before, sending this list of clauses to the Solver() object, we shuffle which might provide more randomness. The solver gives us the solved sudoku pair.
    - It is possible (but rare) that the above set of clauses was non-satisfiable. We then repeat the above steps again by shuffling v[] and rand[] and proceed until we get a pair.
- We store the obtained sudoku to print the solution later. We also make a clause (Temp) containing the negation of all the assigned variables obtained which we use to check if it the sudokus in the further step have a solution distinct from this or not i.e. to check for uniqueness. We add this clause to the original list of clauses (i.e. before generating the random sudoku).

- We all make a list of list (Temp2) which has all the assigned variables individually. We will use this. We shuffle this list. Also note that it has k**4 elements.
- We now need to remove the values at some of the cells from the two sudokus. We iterate through the elements of Temp2
    - From Temp2, we pop one element at a time and check if after removing it, then appending Temp2 to the list of clauses and solving the resulting CNF, we get a possible solution or not.
        - If Solver() returns None then we continue the iteration.
        - If however, there is a distinct sudoku pair possible, we add the popped element back to Temp2 before moving to next iteration.
- After all the iterations above Temp2 contains all the variable assignments needed for the maximal sudoku.
- We print the puzzles to the console and output them to a csv file named “MaximalPair.csv”. We also print their unique solutions to the console and to the file “MaximalPairSolution.csv”
- The Solution can be verified by using Part 1 of the assignment. To verify that the sudoku is unique, one also needs to add an extra clause (Temp) as described above. To verify that the sudoku is maximal, one can edit any of the non-zero cells of the output csv file to zero and run it along with the Temp clause to obtain a distinct solution.

Assumptions:

- The csv files are delimited by comma (‘,’).
- The sudokus generated with the help of random.shuffle() method are sufficiently random.

Limitations:

- The code is extremely slow. For the first part, for k=3 it takes around 1 to 2 seconds to obtain the solution. k=2 case is returned almost instantaneously. We were unable to test cases for k=3 and beyond due to large runtime. This is possibly due to bad encoding of the problem. Our encoding contains O(k**8) clauses.
- The second part is even slower and take around [k**4 times time taken to solve the first part]. This is because we are popping one element at a time and checking for the uniqueness and maximality of the sudoku each time.

How to run:

For the first part:

- Enter the value of k at terminal when prompted. Then enter path (absolute or relative) of the csv file containing the sudokus.

For second part:

- Simply enter the value of k when prompted.