Question 1:

1. Variable: I will treat each cell in a 9x9 Sudoku table as a variable, and denote each variable as S[row][column], where
    - row: row index - {0,1,2,...,8}
    - column: column index - {0,1,2,...,8}
    - There are 9 smaller 3x3 boxes, and each box is denoted as (boxrow, boxcol), where:
        + Boxrow: box's row position - {0,1,2}
        + Boxcol: box's column position - {0,1,2}
        + Each box starts at the grid position:
            - startRow = 3 * boxRow
            - StartCol = 3 * boxCol
2. Domain: {1, 2, 3, 4, 5, 6, 7, 8, 9}
3. Constraints:
    - For all rows in a range of 0 to 8, the values of S[row][0], S[row][1], …, S[row][8] are different from one another
    - For all columns in a range of 0 to 8, S[0][column], S[1][column],..., S[8][column] are different from one another
    - For each 3x3 box, starting at (startRow, startCol), all 9 variables S[startRow+i][startCol+j] | i and j in a range from 0 to 2, are different from one another

Question 2:

a. The Arc-consistency AC-3 algorithm helps detect failures by checking if the domain of a variable is empty, which means there are no possible values left that satisfy the constraints.
   Time complexity of AC-3 for AC-3 is O(cd^3) with:
    - c is the number of constraints or arcs as input
    - Each arc can be pushed back into the queue d times
    - d^2 is the time used in Remove-Inconsistent-Values, as it spends d for going over each value in the domain and d for checking consistency

   Or O(n^2d^3) if each pair has constraint

b. Step 0: We start with
    - Queue = [(X,Y), (Y,X), (Y, Z), (Z,Y), (X, Z), (Z, X)]
    - D(X) = {1,2,3,4}, D(Y) = {1,2,3,4}, D(Z)={1,2,3,4}
    - No new reduced domain of D(i) after enforcing the arc
    - No new arc was added back to the queue

+ Step 1: (X, Y) as X > Y
    - queue= [(Y,X), (Y, Z), (Z,Y), (X, Z), (Z, X)]

- D(X) = {2,3,4}, D(Y) = {1,2,3,4}, D(Z) = {1,2,3,4}
- D(X) got reduced to only 3 values - 1 was removed
- Remove (X,Y) from the queue, check if (Y,X) or (Y,Z) are in the queue or not -> both are already in the queue -> no new arc was added back to the queue

+ Step 2: (Y, X) as X<Y
  - queue = [(Y, Z), (Z,Y), (X, Z), (Z, X)]
  - D(X) = {2,3,4}, D(Y) = {1,2,3}, D(Z) = {1,2,3,4}
  - D(Y) got reduced to only 3 values - 4 was removed
  - Remove (Y,X) from the queue, check if (X,Y) or (X,Z) are in the queue or not
    - (X, Z) already in queue -> not adding it in
    - (X, Y) not in queue (removed in step 1) -> adding it back to the queue => queue right now: = [(Y,Z) (Z,Y), (X, Z), (Z, X), (X,Y)]

+ Step 3: (Y,Z) as Y>Z
  - Queue = [(Z,Y), (X, Z), (Z, X), (X,Y)]
  - D(X) = {2,3,4}, D(Y)= {2,3}, D(Z)={1,2,3,4}
  - D(Y) got reduced to only 2 values - 1 was removed
  - Remove (Y,Z) from the queue, check if (Z,Y) or (Z,X) are in the queue or not -> both are still in the queue -> not adding any new arc into the queue

+ Step 4: (Z,Y) as Z< Y
  - Queue = [(X, Z), (Z, X), (X,Y)]
  - D(X) = {2,3,4}, D(Y)={2,3}, D(Z)={1,2}
  - D(Z) got reduced to only 2 values - 3 and 4 were removed
  - Remove (Z,Y) from the queue, check if (Y,Z) or (Y,X) in queue or not
    - Both are not in queue (removed) -> add both of them back => queue right now = [(X, Z), (Z, X), (X,Y), (Y,Z), (Y,X)]

+ Step 5: (X,Z) as X<Z
  - Queue = [(Z, X), (X,Y), (Y,Z), (Y,X)]
  - D(X)= {}, D(Y)={2,3}, D(Z)={1,2}
  - D(X) is now empty => there are no possible values to satisfy the constraints
  - Failure detected

Question 3:

a. Ordering can impact Alpha-Beta pruning in a minmax tree
   Ex: We have a minmax tree

```
                    MaxM
                /          \
             Min           Min
           /   \          /    \
        MaxA   MaxB    MaxC  MaxD
        /   \    /  \    /  \   /  \
       4     7  2    9  1   6  3   5
```

This is the case where it does not have any specific order

- Left side:

For MaxA: The first value will be 4 before changing to 7
For MaxB: The first value will be 2 before changing to 9
=> Min will be 7 with a=7 back to root

- Right side:

For MaxC: The first value will be 1 before changing to 6
For MaxD: it might choose 5 as max; however, 7> 6, there is no point in going to MaxD -> prune MaxD with nodes 3 and 5
=> MaxM chooses 7 as (final) value, 1 value will be pruned

This is the case where it does have a specific order:

```
                    MaxM
                /          \
             Min           Min
           /   \          /    \
        MaxA   MaxB    MaxC  MaxD
        /   \    /  \    /  \   /  \
       7     4  9    2  6   1  3   5
```

- Left side:

For MaxA: similar to the first situation, 7
For MaxB: it will also be 9, but in this case, since 9 goes before 2, then 2 will be pruned
=> Min will be 7 with a=7 back to root, similar to the first case

- Right side:

For MaxC: the same situation with MaxB, as 6 now will go before 1, so 1 will be pruned
For MaxD: will be pruned, as if MaxM will choose to follow the Left Min path (since 6<7)
=> MaxM still chooses 7 as the final value, but this time, more nodes will be pruned, saving memory

b. Changes to work with a chance-based game are to use Expectiminimax. It's just like minimax, except we have to also handle chance node. This will be very useful in cases where we know exactly what the percentage of a result is from an action.