# Python3 implementation of the

# above approach

from random import randint

N = 8

# A utility function that configures

# the 2D array "board" and

# array "state" randomly to provide

# a starting point for the algorithm.

def configureRandomly(board, state):

# Iterating through the

# column indices

for i in range(N):

# Getting a random row index

state[i] = randint(0, 100000) % N;

# Placing a queen on the

# obtained place in

# chessboard.

board[state[i]][i] = 1;

# A utility function that prints

# the 2D array "board".

def printBoard(board):

for i in range(N):

print(\*board[i])

# A utility function that prints

# the array "state".

def printState( state):

print(\*state)

# A utility function that compares

# two arrays, state1 and state2 and

# returns True if equal

# and False otherwise.

def compareStates(state1, state2):

for i in range(N):

if (state1[i] != state2[i]):

return False;

return True;

# A utility function that fills

# the 2D array "board" with

# values "value"

def fill(board, value):

for i in range(N):

for j in range(N):

board[i][j] = value;

# This function calculates the

# objective value of the

# state(queens attacking each other)

# using the board by the

# following logic.

def calculateObjective( board, state):

# For each queen in a column, we check

# for other queens falling in the line

# of our current queen and if found,

# any, then we increment the variable

# attacking count.

# Number of queens attacking each other,

# initially zero.

attacking = 0;

# Variables to index a particular

# row and column on board.

for i in range(N):

# At each column 'i', the queen is

# placed at row 'state[i]', by the

# definition of our state.

# To the left of same row

# (row remains constant

# and col decreases)

row = state[i]

col = i - 1;

while (col >= 0 and board[row][col] != 1) :

col -= 1

if (col >= 0 and board[row][col] == 1) :

attacking += 1;

# To the right of same row

# (row remains constant

# and col increases)

row = state[i]

col = i + 1;

while (col < N and board[row][col] != 1):

col += 1;

if (col < N and board[row][col] == 1) :

attacking += 1;

# Diagonally to the left up

# (row and col simultaneously

# decrease)

row = state[i] - 1

col = i - 1;

while (col >= 0 and row >= 0 and board[row][col] != 1) :

col-= 1;

row-= 1;

if (col >= 0 and row >= 0 and board[row][col] == 1) :

attacking+= 1;

# Diagonally to the right down

# (row and col simultaneously

# increase)

row = state[i] + 1

col = i + 1;

while (col < N and row < N and board[row][col] != 1) :

col+= 1;

row+= 1;

if (col < N and row < N and board[row][col] == 1) :

attacking += 1;

# Diagonally to the left down

# (col decreases and row

# increases)

row = state[i] + 1

col = i - 1;

while (col >= 0 and row < N and board[row][col] != 1) :

col -= 1;

row += 1;

if (col >= 0 and row < N and board[row][col] == 1) :

attacking += 1;

# Diagonally to the right up

# (col increases and row

# decreases)

row = state[i] - 1

col = i + 1;

while (col < N and row >= 0 and board[row][col] != 1) :

col += 1;

row -= 1;

if (col < N and row >= 0 and board[row][col] == 1) :

attacking += 1;

# Return pairs.

return int(attacking / 2);

# A utility function that

# generates a board configuration

# given the state.

def generateBoard( board, state):

fill(board, 0);

for i in range(N):

board[state[i]][i] = 1;

# A utility function that copies

# contents of state2 to state1.

def copyState( state1, state2):

for i in range(N):

state1[i] = state2[i];

# This function gets the neighbour

# of the current state having

# the least objective value

# amongst all neighbours as

# well as the current state.

def getNeighbour(board, state):

# Declaring and initializing the

# optimal board and state with

# the current board and the state

# as the starting point.

opBoard = [[0 for \_ in range(N)] for \_ in range(N)]

opState = [0 for \_ in range(N)]

copyState(opState, state);

generateBoard(opBoard, opState);

# Initializing the optimal

# objective value

opObjective = calculateObjective(opBoard, opState);

# Declaring and initializing

# the temporary board and

# state for the purpose

# of computation.

NeighbourBoard = [[0 for \_ in range(N)] for \_ in range(N)]

NeighbourState = [0 for \_ in range(N)]

copyState(NeighbourState, state);

generateBoard(NeighbourBoard, NeighbourState);

# Iterating through all

# possible neighbours

# of the board.

for i in range(N):

for j in range(N):

# Condition for skipping the

# current state

if (j != state[i]) :

# Initializing temporary

# neighbour with the

# current neighbour.

NeighbourState[i] = j;

NeighbourBoard[NeighbourState[i]][i] = 1;

NeighbourBoard[state[i]][i] = 0;

# Calculating the objective

# value of the neighbour.

temp = calculateObjective( NeighbourBoard, NeighbourState);

# Comparing temporary and optimal

# neighbour objectives and if

# temporary is less than optimal

# then updating accordingly.

if (temp <= opObjective) :

opObjective = temp;

copyState(opState, NeighbourState);

generateBoard(opBoard, opState);

# Going back to the original

# configuration for the next

# iteration.

NeighbourBoard[NeighbourState[i]][i] = 0;

NeighbourState[i] = state[i];

NeighbourBoard[state[i]][i] = 1;

# Copying the optimal board and

# state thus found to the current

# board and, state since c+= 1 doesn't

# allow returning multiple values.

copyState(state, opState);

fill(board, 0);

generateBoard(board, state);

def hillClimbing(board, state):

# Declaring and initializing the

# neighbour board and state with

# the current board and the state

# as the starting point.

neighbourBoard = [[0 for \_ in range(N)] for \_ in range(N)]

neighbourState = [0 for \_ in range(N)]

copyState(neighbourState, state);

generateBoard(neighbourBoard, neighbourState);

while True:

# Copying the neighbour board and

# state to the current board and

# state, since a neighbour

# becomes current after the jump.

copyState(state, neighbourState);

generateBoard(board, state);

# Getting the optimal neighbour

getNeighbour(neighbourBoard, neighbourState);

if (compareStates(state, neighbourState)) :

# If neighbour and current are

# equal then no optimal neighbour

# exists and therefore output the

# result and break the loop.

printBoard(board);

break;

elif (calculateObjective(board, state) == calculateObjective( neighbourBoard,neighbourState)):

# If neighbour and current are

# not equal but their objectives

# are equal then we are either

# approaching a shoulder or a

# local optimum, in any case,

# jump to a random neighbour

# to escape it.

# Random neighbour

neighbourState[randint(0, 100000) % N] = randint(0, 100000) % N;

generateBoard(neighbourBoard, neighbourState);

# Driver code

state = [0] \* N

board = [[0 for \_ in range(N)] for \_ in range(N)]

# Getting a starting point by

# randomly configuring the board

configureRandomly(board, state);

# Do hill climbing on the

# board obtained

hillClimbing(board, state);

# This code is contributed by phasing17.

Output

0 0 1 0 0 0 0 0

0 0 0 0 0 1 0 0

0 0 0 0 0 0 0 1

1 0 0 0 0 0 0 0

0 0 0 1 0 0 0 0

0 0 0 0 0 0 1 0

0 0 0 0 1 0 0 0

0 1 0 0 0 0 0 0