```
# NAME : VARSHA VALECHA
# ROLL NO : A 26


from random import choice
from math import inf

board = [[0, 0, 0],
         [0, 0, 0],
         [0, 0, 0]]

def Gameboard(board):
    chars = {1: 'X', -1: 'O', 0: ' '}
    for x in board:
        for y in x:
            ch = chars[y]
            print(f'| {ch} |', end='')
        print('\n' + '---------------')
    print('===============')

def Clearboard(board):
    for x, row in enumerate(board):
        for y, col in enumerate(row):
            board[x][y] = 0

def winningPlayer(board, player):
    conditions = [[board[0][0], board[0][1], board[0][2]],
                  [board[1][0], board[1][1], board[1][2]],
                  [board[2][0], board[2][1], board[2][2]],
                  [board[0][0], board[1][0], board[2][0]],
                  [board[0][1], board[1][1], board[2][1]],
                  [board[0][2], board[1][2], board[2][2]],
                  [board[0][0], board[1][1], board[2][2]],
                  [board[0][2], board[1][1], board[2][0]]]

    if [player, player, player] in conditions:
        return True

    return False

def gameWon(board):
    return winningPlayer(board, 1) or winningPlayer(board, -1)

def printResult(board):
    if winningPlayer(board, 1):
        print('X has won! ' + '\n')

    elif winningPlayer(board, -1):
        print('O\'s have won! ' + '\n')

    else:
        print('Draw' + '\n')

def blanks(board):
    blank = []
    for x, row in enumerate(board):
        for y, col in enumerate(row):
            if board[x][y] == 0:
                blank.append([x, y])

    return blank

def boardFull(board):
    if len(blanks(board)) == 0:
        return True
    return False

def setMove(board, x, y, player):
    board[x][y] = player

def playerMove(board):
    e = True
    moves = {1: [0, 0], 2: [0, 1], 3: [0, 2],
             4: [1, 0], 5: [1, 1], 6: [1, 2],
             7: [2, 0], 8: [2, 1], 9: [2, 2]}
    while e:
        try:
            move = int(input('Enter a number between 1-9: '))
            if move < 1 or move > 9:
                print('Invalid Move! Try again!')
            elif not (moves[move] in blanks(board)):
                print('Invalid Move! Try again!')
```

```
                print('Invalid Move! Try again!')
            else:
                setMove(board, moves[move][0], moves[move][1], 1)
                Gameboard(board)
                e = False
        except(KeyError, ValueError):
            print('Enter a number!')

def getScore(board):
    if winningPlayer(board, 1):
        return 10

    elif winningPlayer(board, -1):
        return -10

    else:
        return 0

def abminimax(board, depth, alpha, beta, player):
    row = -1
    col = -1
    if depth == 0 or gameWon(board):
        return [row, col, getScore(board)]

    else:
        for cell in blanks(board):
            setMove(board, cell[0], cell[1], player)
            score = abminimax(board, depth - 1, alpha, beta, -player)
            if player == 1:
                # X is always the max player
                if score[2] > alpha:
                    alpha = score[2]
                    row = cell[0]
                    col = cell[1]

            else:
                if score[2] < beta:
                    beta = score[2]
                    row = cell[0]
                    col = cell[1]

            setMove(board, cell[0], cell[1], 0)

            if alpha >= beta:
                break

        if player == 1:
            return [row, col, alpha]

        else:
            return [row, col, beta]

def o_comp(board):
    if len(blanks(board)) == 9:
        x = choice([0, 1, 2])
        y = choice([0, 1, 2])
        setMove(board, x, y, -1)
        Gameboard(board)

    else:
        result = abminimax(board, len(blanks(board)), -inf, inf, -1)
        setMove(board, result[0], result[1], -1)
        Gameboard(board)

def x_comp(board):
    if len(blanks(board)) == 9:
        x = choice([0, 1, 2])
        y = choice([0, 1, 2])
        setMove(board, x, y, 1)
        Gameboard(board)

    else:
        result = abminimax(board, len(blanks(board)), -inf, inf, 1)
        setMove(board, result[0], result[1], 1)
        Gameboard(board)

def makeMove(board, player, mode):
    if mode == 1:
        if player == 1:
            playerMove(board)

        else:
            o_comp(board)
```

```
        else:
            if player == 1:
                o_comp(board)
            else:
                x_comp(board)

def pvc():
    while True:
        try:
            order = int(input('Enter to play 1st or 2nd: '))
            if not (order == 1 or order == 2):
                print('Please pick 1 or 2')
            else:
                break
        except(KeyError, ValueError):
            print('Enter a number')

    Clearboard(board)
    if order == 2:
        currentPlayer = -1
    else:
        currentPlayer = 1

    while not (boardFull(board) or gameWon(board)):
        makeMove(board, currentPlayer, 1)
        currentPlayer *= -1

    printResult(board)

# Driver Code
print("================================================")
print("TIC-TAC-TOE using MINIMAX with ALPHA-BETA Pruning")
print("================================================")
pvc()
```

```
---------------
|   ||   ||   |
---------------
===============
| O ||   ||   |
---------------
|   || X ||   |
---------------
|   ||   ||   |
---------------
===============
Enter a number between 1-9: 6
| O ||   ||   |
---------------
|   || X || X |
---------------
|   ||   ||   |
---------------
===============
| O ||   ||   |
---------------
| O || X || X |
---------------
|   ||   ||   |
---------------
===============
Enter a number between 1-9: 7
| O ||   ||   |
---------------
| O || X || X |
---------------
| X ||   ||   |
---------------
===============
| O ||   || O |
---------------
| O || X || X |
---------------
| X ||   ||   |
---------------
===============
Enter a number between 1-9: 9
| O ||   || O |
---------------
```

```
---------------
| X ||   || X |
---------------
==============
O's have won!
```

```
#simple minimax

# A simple Python3 program to find
# maximum score that
# maximizing player can get
import math

def minimax (curDepth, nodeIndex,
            maxTurn, scores,
            targetDepth):

    # base case : targetDepth reached
    if (curDepth == targetDepth):
        return scores[nodeIndex]

    if (maxTurn):
        return max(minimax(curDepth + 1, nodeIndex * 2,
                    False, scores, targetDepth),
                minimax(curDepth + 1, nodeIndex * 2 + 1,
                    False, scores, targetDepth))

    else:
        return min(minimax(curDepth + 1, nodeIndex * 2,
                    True, scores, targetDepth),
                minimax(curDepth + 1, nodeIndex * 2 + 1,
                    True, scores, targetDepth))

# Driver code
scores = [3, 5, 2, 9, 12, 5, 23, 23]

treeDepth = math.log(len(scores), 2)

print("The optimal value is : ", end = "")
print(minimax(0, 0, True, scores, treeDepth))

# This code is contributed
# by rootshadow
```

```
# minimax with alpha beta bruning


# working of Alpha-Beta Pruning

# Initial values of Alpha and Beta
MAX, MIN = 1000, -1000

# Returns optimal value for current player
#(Initially called for root and maximizer)
def minimax(depth, nodeIndex, maximizingPlayer,
            values, alpha, beta):

    # Terminating condition. i.e
    # leaf node is reached
    if depth == 3:
        return values[nodeIndex]

    if maximizingPlayer:

        best = MIN

        # Recur for left and right children
        for i in range(0, 2):

            val = minimax(depth + 1, nodeIndex * 2 + i,
                        False, values, alpha, beta)
            best = max(best, val)
            alpha = max(alpha, best)

            # Alpha Beta Pruning
            if beta <= alpha:
                break

        return best
```

```
    else:
        best = MAX

        # Recur for left and
        # right children
        for i in range(0, 2):

            val = minimax(depth + 1, nodeIndex * 2 + i,
                            True, values, alpha, beta)
            best = min(best, val)
            beta = min(beta, best)

            # Alpha Beta Pruning
            if beta <= alpha:
                break

        return best

# Driver Code
if __name__ == "__main__":

    values = [3, 5, 6, 9, 1, 2, 0, -1]
    print("The optimal value is :", minimax(0, 0, True, values, MIN, MAX))
```

```
# Python3 program to find the next optimal move for a player
player, opponent = 'x', 'o'

# This function returns true if there are moves
# remaining on the board. It returns false if
# there are no moves left to play.
def isMovesLeft(board) :

  for i in range(3) :
    for j in range(3) :
      if (board[i][j] == '_') :
        return True
  return False

# This is the evaluation function as discussed
# in the previous article ( http://goo.gl/sJgv68 )
def evaluate(b) :

  # Checking for Rows for X or O victory.
  for row in range(3) :
    if (b[row][0] == b[row][1] and b[row][1] == b[row][2]) :
      if (b[row][0] == player) :
        return 10
      elif (b[row][0] == opponent) :
        return -10

  # Checking for Columns for X or O victory.
  for col in range(3) :

    if (b[0][col] == b[1][col] and b[1][col] == b[2][col]) :

      if (b[0][col] == player) :
        return 10
      elif (b[0][col] == opponent) :
        return -10

  # Checking for Diagonals for X or O victory.
  if (b[0][0] == b[1][1] and b[1][1] == b[2][2]) :

    if (b[0][0] == player) :
      return 10
    elif (b[0][0] == opponent) :
      return -10

  if (b[0][2] == b[1][1] and b[1][1] == b[2][0]) :

    if (b[0][2] == player) :
      return 10
    elif (b[0][2] == opponent) :
      return -10

  # Else if none of them have won then return 0
  return 0

# This is the minimax function. It considers all
# the possible ways the game can go and returns
```

```
# the value of the board
def minimax(board, depth, isMax) :
  score = evaluate(board)

  # If Maximizer has won the game return his/her
  # evaluated score
  if (score == 10) :
    return score

  # If Minimizer has won the game return his/her
  # evaluated score
  if (score == -10) :
    return score

  # If there are no more moves and no winner then
  # it is a tie
  if (isMovesLeft(board) == False) :
    return 0

  # If this maximizer's move
  if (isMax) :
    best = -1000

    # Traverse all cells
    for i in range(3) :
      for j in range(3) :

        # Check if cell is empty
        if (board[i][j]=='_') :

          # Make the move
          board[i][j] = player

          # Call minimax recursively and choose
          # the maximum value
          best = max( best, minimax(board,
                      depth + 1,
                      not isMax) )

          # Undo the move
          board[i][j] = '_'
    return best

  # If this minimizer's move
  else :
    best = 1000

    # Traverse all cells
    for i in range(3) :
      for j in range(3) :

        # Check if cell is empty
        if (board[i][j] == '_') :

          # Make the move
          board[i][j] = opponent

          # Call minimax recursively and choose
          # the minimum value
          best = min(best, minimax(board, depth + 1, not isMax))

          # Undo the move
          board[i][j] = '_'
    return best

# This will return the best possible move for the player
def findBestMove(board) :
  bestVal = -1000
  bestMove = (-1, -1)

  # Traverse all cells, evaluate minimax function for
  # all empty cells. And return the cell with optimal
  # value.
  for i in range(3) :
    for j in range(3) :

      # Check if cell is empty
      if (board[i][j] == '_') :

        # Make the move
        board[i][j] = player
```

```
        # compute evaluation function for this
        # move.
        moveVal = minimax(board, 0, False)

        # Undo the move
        board[i][j] = '_'

        # If the value of the current move is
        # more than the best value, then update
        # best/
        if (moveVal > bestVal) :
          bestMove = (i, j)
          bestVal = moveVal

  print("The value of the best Move is :", bestVal)
  print()
  return bestMove
# Driver code
board = [
  [ 'x', 'o', 'x' ],
  [ 'o', 'o', 'x' ],
  [ '_', '_', '_' ]
]

bestMove = findBestMove(board)

print("The Optimal Move is :")
print("ROW:", bestMove[0], " COL:", bestMove[1])

# This code is contributed by divyesh072019
```