EAST WEST UNIVERSITY

Project Report

Project No - 3
Project Title:
Develop a two-opponent chess game using necessary algorithms and implement all possible cases of the n-queens algorithm as well.

Submitted By:

Name	ID
Kasif Hasnaen Zisan	2020-1-60-028
Rifa Tasnim Orin	2020-1-60-103
Bahauddin Ahmed	2020-1-60-271
Md. Golam Muhit	2020-1-60-269
Mashrufa Khandaker	2020-1-60-030

Submitted To:

Dr. Taskeed Jabid
Chairperson, Associate Professor
Department of Computer Science & Engineering
East West University

Date of Submission:

18/9/2022

Project Title:
Develop a two-opponent chess game using necessary algorithms and implement all possible cases of the n-queens algorithm as well.

Introduction:
Our project was to build a two player chess game using necessary algorithms and implement all possible cases of the n- queens algorithm.
In this project report, we will discuss our solution of the given project, our code implementation, the decisions we made to complete this project and also some future prospects of the project.

Programming Language:
- Python

Libraries Used:
- pygame

Prerequisites:
To run this project, one has to have pygame installed in their local machine. To install pygame, just simply type in your terminal - `pip install pygame`
Also, all of the files (two python files and the image file) have to be in the same folder, for the project to be run.

Features Implemented:
1. Chessboard with GUI.
2. N-Queens on the chessboard using the N-Queens backtracking algorithm.
3. Movement of the chess pieces on the board using mouse clicks.
4. Valid move generation for all the chess pieces.
5. Incorrect move warning if any incorrect move is made.
6. Warning if a side attacks their own chess piece.
7. Simple win or lose decision making based on if the king still exists.

Design Decisions:
- **Python -** We used python as our programming language because python makes implementation of complex logics very simple. Also as this is a game, we wanted to make use of the pygame library of python to implement a simple GUI for this project.
- **pygame -** We used pygame to create the GUI for our Chess game.
- **Keeping it Simple -** Chess is a complex game to make. But as this is a mini project and had to be done in limited time, we did not have enough time to implement all the cases for our game. Our project game assumes that the two players who are playing know the

rules of the chess game, so that they can make some decisions themselves while playing. Some of the design decisions we took while making this game was -

- **Not Implementing a Checkmate System** because checkmate condition implementation is very complex. Before the program moves a chess piece of a certain side, it has to check all the possible moves of the other side to check if the movement of this chess piece will cause a checkmate or not. So checking all the possible moves of the other side becomes a complex task, which is hard to implement in a short time. Instead of this we implemented a simple win or lose system.
- **Not implementing a turn based playing system.** Our chess game assumes that the two players playing the game will maintain their turn on their own. We wanted to implement this feature but implementation of this will cause our project structure to change so we did not implement it at the last moment.
- **Not implementing an undo method.** We could not implement an undo method because of the limited time.

Future Prospects of the Project

- Implementation of all of the chess game cases
- Implementation of checkmate condition
- Implementation of turn based playing system
- Implementation of an undo method and creating a move log containing all the previous moves.
- Implementation of a simple AI system.
- Tile highlighting to indicate which tile a certain piece can move when selected.

Code-

Our project contains two python files and one image file containing (.PNG) images of all of the chess pieces.

The Two files are **`main.py`** and **`NQueensBoard.py.`**

`main.py -`

```
import pygame as p

from Chess import NQueensBoard

WIDTH = HEIGHT = 512
DIMENSION = 8
SQ_SIZE = HEIGHT // DIMENSION
MAX_FPS = 15
IMAGES = { }


def loadImages( ):
   pieces = ['bR', 'bN', 'bB', 'bQ', 'bK', 'bP', 'wP', 'wR', 'wN', 'wB', 'wQ', 'wK']
   for piece in pieces:
        IMAGES[piece] = p.transform.scale(p.image.load("images/" + piece + ".png"), (SQ_SIZE,
SQ_SIZE))


def main( ):
   p.init( )
   screen = p.display.set_mode((WIDTH, HEIGHT))
   clock = p.time.Clock()
   screen.fill(p.Color("white"))
   NQueensBoard.N_queens(8)
   loadImages()
   running = True
   sqSelected = ()
   playerClicks = []
   while running:
      for e in p.event.get():
         if e.type == p.QUIT:
            running = False
         elif e.type == p.MOUSEBUTTONDOWN:
            location = p.mouse.get_pos()
            col = location[0] // SQ_SIZE
            row = location[1] // SQ_SIZE
            sqSelected = (row, col)
            playerClicks.append(sqSelected)
            if len(playerClicks) == 2:
```

```
                if NQueensBoard.isValid(playerClicks[0], playerClicks[1]):
                    NQueensBoard.move(playerClicks[0], playerClicks[1])
                    sqSelected = ()
                    playerClicks = []
                else:
                    print("Invalid Move, Please give correct move")
                    playerClicks[1] = playerClicks[0]
                    NQueensBoard.move(playerClicks[0], playerClicks[1])
                    sqSelected = ()
                    playerClicks = []


        drawGameState(screen)
        clock.tick(MAX_FPS)
        p.display.flip()


def drawGameState(screen):
    drawBoard(screen)
    drawPieces(screen, NQueensBoard.board)


def drawBoard(screen):
    colors = [p.Color("white"), p.Color("gray")]
    for r in range(DIMENSION):
        for c in range(DIMENSION):
            color = colors[((r + c) % 2)]
            p.draw.rect(screen, color, p.Rect(c * SQ_SIZE, r * SQ_SIZE, SQ_SIZE, SQ_SIZE))


def drawPieces(screen, board):
    for r in range(DIMENSION):
        for c in range(DIMENSION):
            piece = board[r][c]
            if piece != "--":
                screen.blit(IMAGES[piece], p.Rect(c * SQ_SIZE, r * SQ_SIZE, SQ_SIZE, SQ_SIZE))


if __name__ == "__main__":
    main()
```

NQueensBoard.py –

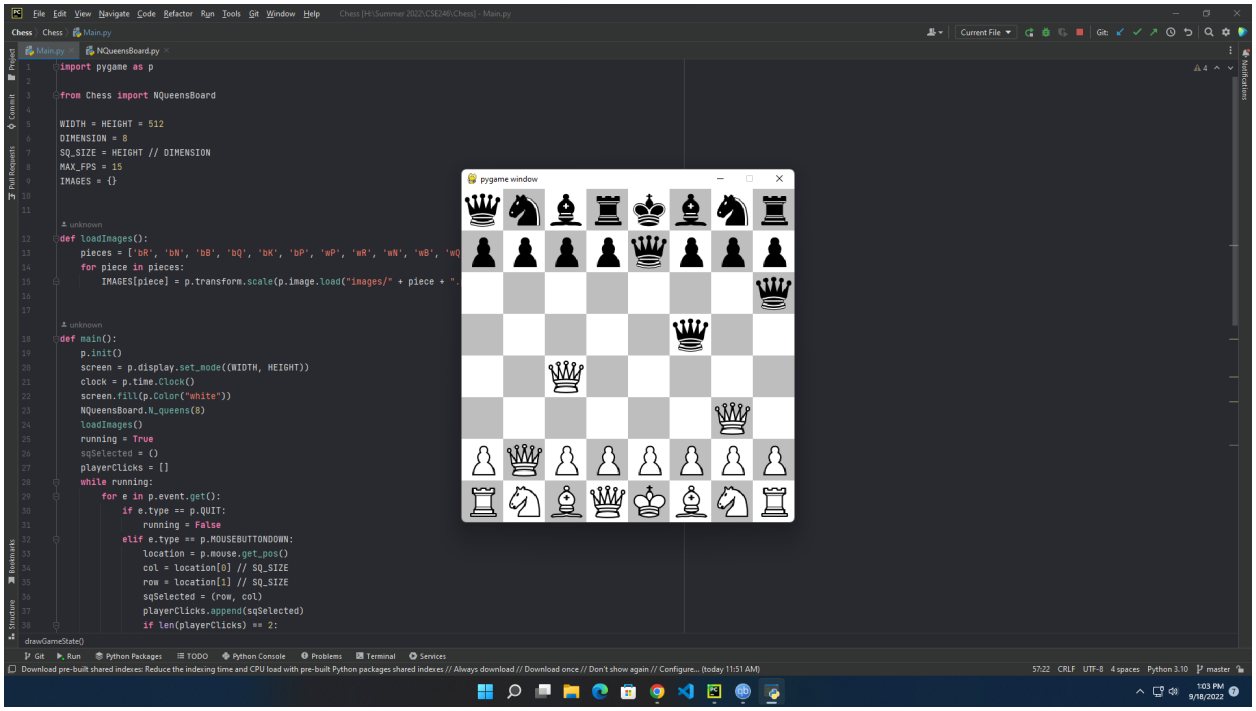
```
global N

N = 8
board = [
   ['bR', 'bN', 'bB', 'bR', 'bK', 'bB', 'bN', 'bR'],
   ['bP', 'bP', 'bP', 'bP', 'bP', 'bP', 'bP', 'bP'],
   ['--', '--', '--', '--', '--', '--', '--', '--'],
   ['--', '--', '--', '--', '--', '--', '--', '--'],
   ['--', '--', '--', '--', '--', '--', '--', '--'],
   ['--', '--', '--', '--', '--', '--', '--', '--'],
   ['wP', 'wP', 'wP', 'wP', 'wP', 'wP', 'wP', 'wP'],
   ['wR', 'wN', 'wB', '--', 'wK', 'wB', 'wN', 'wR'],
]

whiteToMove = True


def attack(i, j):
   for k in range(0, N):
      if board[i][k] == 'bQ' or board[i][k] == 'wQ' or board[k][j] == 'bQ' or board[k][j] == 'wQ':
         return True
   for k in range(0, N):
      for l in range(0, N):
         if (k + l == i + j) or (k - l == i - j):
            if board[k][l] == 'bQ' or board[k][l] == 'wQ':
               return True
   return False


def N_queens(n):
   if n == 0:
      return True
   for i in range(0, N):
      for j in range(0, N):
         if (not (attack(i, j))) and (board[i][j] != 'wQ' or board[i][j] != 'bQ'):
            save = board[i][j]
            if i <= 3:
               board[i][j] = 'bQ'
            elif i > 3:
               board[i][j] = 'wQ'
            if N_queens(n - 1):
               return True
            board[i][j] = save
   return False
```

```
def move(startSq, endSq):
    startRow = startSq[0]
    startCol = startSq[1]
    endRow = endSq[0]
    endCol = endSq[1]
    pieceCaptured = board[endRow][endCol]
    pieceMoved = board[startRow][startCol]
    movedColor = board[startRow][startCol][0]
    capturedColor = board[endRow][endCol][0]
    if movedColor != '-' and capturedColor != '-' and startSq != endSq and movedColor ==
capturedColor:
        print("You cannot attack your own")
    else:
        board[startRow][startCol] = '--'
        board[endRow][endCol] = pieceMoved
        if pieceCaptured == 'wK':
            print("BLACK HAS WON")
        elif pieceCaptured == 'bK':
            print("WHITE HAS WON")

def isValid(startSq, endSq):
    validFlag = True
    startRow = startSq[0]
    startCol = startSq[1]
    endRow = endSq[0]
    endCol = endSq[1]
    piece = board[startRow][startCol][1]
    pieceColor = board[startRow][startCol][0]
    if piece == 'P':
        if not movePawn(startRow, startCol, endRow, endCol, pieceColor):
            validFlag = False
    elif piece == 'R':
        if not moveRook(startRow, startCol, endRow, endCol):
            validFlag = False
    elif piece == 'N':
        if not moveKnight(startRow, startCol, endRow, endCol):
            validFlag = False
    elif piece == 'B':
        if not moveBishop(startRow, startCol, endRow, endCol):
            validFlag = False
    elif piece == 'Q':
        if not moveQueen(startRow, startCol, endRow, endCol):
            validFlag = False
    elif piece == 'K':
```

```
        if not moveKing(startRow, startCol, endRow, endCol):
            validFlag = False
    if validFlag:
        return True
    else:
        return False


def movePawn(startRow, startCol, endRow, endCol, pieceColor):
    if pieceColor == 'b' and startRow == 1 and ((startRow < endRow <= startRow + 2) or (
            startRow < endRow and (endCol == startCol + 1 or endCol == startCol - 1))):
        return True
    elif pieceColor == 'b' and startRow != 1 and ((startRow < endRow == startRow + 1) or (
            startRow < endRow and (endCol == startCol + 1 or endCol == startCol - 1))):
        return True
    elif pieceColor == 'w' and startRow == 6 and ((startRow > endRow >= startRow - 2) or (
            startRow > endRow and (endCol == startCol + 1 or endCol == startCol - 1))):
        return True
    elif pieceColor == 'w' and startRow != 6 and ((startRow > endRow == startRow - 1) or (
            startRow > endRow and (endCol == startCol + 1 or endCol == startCol - 1))):
        return True

    return False


def moveRook(startRow, startCol, endRow, endCol):
    if startCol != endCol and startRow != endRow:
        return False
    else:
        return True


def moveKnight(startRow, startCol, endRow, endCol):
     if (endCol == startCol + 1 or endCol == startCol - 1) and (endRow == startRow - 2 or endRow
== startRow + 2):
        return True
       elif (endCol == startCol + 2 or endCol == startCol - 2) and (endRow == startRow - 1 or
endRow == startRow + 1):
        return True
    else:
        return False


def moveBishop(startRow, startCol, endRow, endCol):
    if endCol > startCol and endRow > startRow:
        colMoved = endCol - startCol
```

```
        rowMoved = endRow - startRow
        if colMoved == rowMoved:
            return True
    elif endCol < startCol and endRow > startRow:
        colMoved = startCol - endCol
        rowMoved = endRow - startRow
        if colMoved == rowMoved:
            return True
    elif endCol < startCol and endRow < startRow:
        colMoved = startCol - endCol
        rowMoved = startRow - endRow
        if colMoved == rowMoved:
            return True
    elif endCol > startCol and endRow < startRow:
        colMoved = endCol - startCol
        rowMoved = startRow - endRow
        if colMoved == rowMoved:
            return True
    else:
        return False


def moveQueen(startRow, startCol, endRow, endCol):
        if moveBishop(startRow, startCol, endRow, endCol) or moveRook(startRow, startCol, endRow, endCol):
        return True
    else:
        return False

def moveKing(startRow, startCol, endRow, endCol):
    if endRow == startRow + 1 or endRow == startRow - 1 or endCol == startCol + 1 or endCol == startCol - 1:
        return True
    else:
        return False
```

Code Output:
This is what the output looks like initially when we run the code -



As you can see our code outputs a 8*8 chess board, with 8 queens, 4 for each side. And now this game can be played using mouse clicks.

Code Discussion:
Now we will give a function by function explanation of our code.
We will start with the main.py file.

main.py -

```
import pygame as p

from Chess import NQueensBoard

WIDTH = HEIGHT = 512
DIMENSION = 8
SQ_SIZE = HEIGHT // DIMENSION
MAX_FPS = 15
IMAGES = {}
```

Here we are just importing our pygame library and also importing the other file so that we can call the functions from the other file.
We are also setting the height, width and dimension of our GUI chess board and also determining the square size. MAX_FPS was for animation if we implemented it.
And then we are also initializing a dictionary called IMAGES where we can keep our chess piece images.

def loadImages():

```
    unknown
12  def loadImages():
13      pieces = ['bR', 'bN', 'bB', 'bQ', 'bK', 'bP', 'wP', 'wR', 'wN', 'wB', 'wQ', 'wK']
14      for piece in pieces:
15          IMAGES[piece] = p.transform.scale(p.image.load("images/" + piece + ".png"), (SQ_SIZE, SQ_SIZE))
16
17
```

In this function, we are initializing a list called pieces. We are keeping the string notation of all the pieces in that list. After that we are filling up the IMAGES dictionary with the pieces as the key and the image of that piece as the value.

```
unknown
def main():
    p.init()
    screen = p.display.set_mode((WIDTH, HEIGHT))
    clock = p.time.Clock()
    screen.fill(p.Color("white"))
    NQueensBoard.N_queens(8)
    loadImages()
    running = True
    sqSelected = ()
    playerClicks = []
    while running:
        for e in p.event.get():
            if e.type == p.QUIT:
                running = False
            elif e.type == p.MOUSEBUTTONDOWN:
                location = p.mouse.get_pos()
                col = location[0] // SQ_SIZE
                row = location[1] // SQ_SIZE
                sqSelected = (row, col)
                playerClicks.append(sqSelected)
                if len(playerClicks) == 2:
                    if NQueensBoard.isValid(playerClicks[0], playerClicks[1]):
                        NQueensBoard.move(playerClicks[0], playerClicks[1])
                        sqSelected = ()
                        playerClicks = []
                    else:
                        print("Invalid Move, Please give correct move")
                        playerClicks[1] = playerClicks[0]
                        NQueensBoard.move(playerClicks[0], playerClicks[1])
                        sqSelected = ()
                        playerClicks = []


        drawGameState(screen)
        clock.tick(MAX_FPS)
        p.display.flip()


```

This is the main method and it is responsible for creating the gamestate, updating the gamestate, validating the moves and moving the chess pieces.

- Line 19 - 22: Here at first we initialize our board accordingly.
- Line 23: We call the Nqueens method from the NQueensBoard file to create our board.
- Line 24: After our board has been created and all the chess pieces are placed on the board, we call the loadImages function to load the images of the pieces on the board.
- Line 26, 27 - We create a tuple sqSelected and a list playerClicks. The tuple will be used to store the row and column number of the chess piece that the player has clicked on. The

list will be used to store the row and column of the selected piece and also the row and column where the player wants to move the piece.

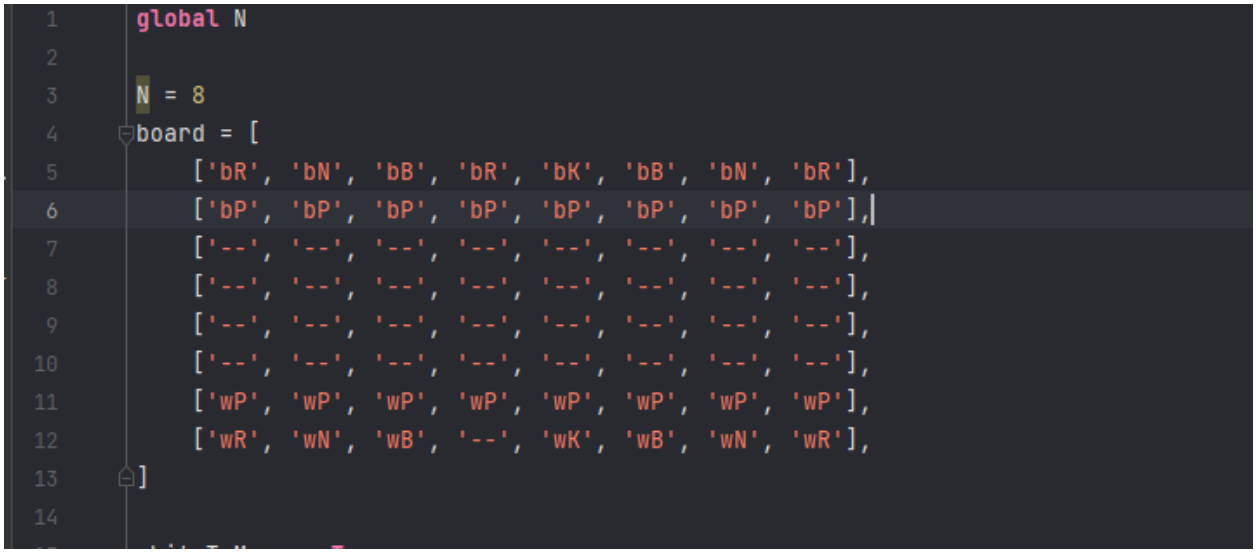
- Line 32 - 48: Here when a player clicks on a piece, its row, column is stored in sqSelected and this tuple is then appended into playerClicks list. Then the program checks if the player has clicked on any other tile, where the player wants to put the chess piece, so if the player does that, the length of the playerClicks list becomes true and the control flow of the program enters the if condition. After that the program checks if the move that was made by the player is valid or not by calling the isValid function. And if the move is valid then the program makes the move. If the move is not valid, then the program prints out a message saying that "Incorrect Move. Please enter the correct move." Whether the move is valid or not, the program clears the tuple and the list, so that it can take the next move.
- Line 51 - 53: Here we are displaying our created board on the screen.

```
56  def drawGameState(screen):
57      drawBoard(screen)
58      drawPieces(screen, NQueensBoard.board)
59
60
        unknown
61  def drawBoard(screen):
62      colors = [p.Color("white"), p.Color("gray")]
63      for r in range(DIMENSION):
64          for c in range(DIMENSION):
65              color = colors[((r + c) % 2)]
66              p.draw.rect(screen, color, p.Rect(c * SQ_SIZE, r * SQ_SIZE, SQ_SIZE, SQ_SIZE))
67
68
        unknown
69  def drawPieces(screen, board):
70      for r in range(DIMENSION):
71          for c in range(DIMENSION):
72              piece = board[r][c]
73              if piece != "--":
74                  screen.blit(IMAGES[piece], p.Rect(c * SQ_SIZE, r * SQ_SIZE, SQ_SIZE, SQ_SIZE))
75
76
```

In these three functions, we are creating the board, putting the pieces on to the board, and updating the game state accordingly after each move.

NQueensBoard.py

This file is responsible for the board, implementing the N-Queens algorithm, running the isValid and move function that we called from the main.py file. It also stores the logic for the moves of all the chess pieces.



```
global N

N = 8
board = [
    ['bR', 'bN', 'bB', 'bR', 'bK', 'bB', 'bN', 'bR'],
    ['bP', 'bP', 'bP', 'bP', 'bP', 'bP', 'bP', 'bP'],
    ['--', '--', '--', '--', '--', '--', '--', '--'],
    ['--', '--', '--', '--', '--', '--', '--', '--'],
    ['--', '--', '--', '--', '--', '--', '--', '--'],
    ['--', '--', '--', '--', '--', '--', '--', '--'],
    ['wP', 'wP', 'wP', 'wP', 'wP', 'wP', 'wP', 'wP'],
    ['wR', 'wN', 'wB', '--', 'wK', 'wB', 'wN', 'wR'],
]
```

Here we are just initializing the board using the string notation of the chess pieces. And in the main function, the images of the chess pieces are set on the board and loaded onto the screen accordingly.

Here we can change the value of the N variable to change the number of queens on the board.

```
unknown *
def attack(i, j):
    for k in range(0, N):
        if board[i][k] == 'bQ' or board[i][k] == 'wQ' or board[k][j] == 'bQ' or board[k][j] == 'wQ':
            return True
    for k in range(0, N):
        for l in range(0, N):
            if (k + l == i + j) or (k - l == i - j):
                if board[k][l] == 'bQ' or board[k][l] == 'wQ':
                    return True
    return False


unknown
def N_queens(n):
    if n == 0:
        return True
    for i in range(0, N):
        for j in range(0, N):
            if (not (attack(i, j))) and (board[i][j] != 'wQ' or board[i][j] != 'bQ'):
                save = board[i][j]
                if i <= 3:
                    board[i][j] = 'bQ'
                elif i > 3:
                    board[i][j] = 'wQ'
                if N_queens(n - 1):
                    return True
                board[i][j] = save
    return False
```

def N_queens(n):

Here we are implementing the n queens algorithm. The N-queens algorithm has three cases -

- No more than one queen in a single row.
- No more than one queen in a single column.
- No two queens attack each other initially.

We implement all of these cases in our algorithm.

First we are taking the number of queens, n. Then we are traversing the board using a nested two dimensional loop. Then we check that, if we put a queen in a certain (row, col) position, will it get attacked or not using the attack(row, col) function. If it does not get attacked and the position we want to put the queen in is not already taken by any other queen then we put the queen in that position. If the row is less than or equal to 3 then we put a black queen or else we put a white queen. Before putting the queen we also save the piece that was in that place. So that if later on we find out that putting the queen in this position does not give us a solution, then we can restore the original piece in that position.

def attack(i, j):

Here we check that if we place the queen in this place, will it be attacked by any other queen or not. If it is attacked, we return true, otherwise we return false.

```
unknown
def isValid(startSq, endSq):
    validFlag = True
    startRow = startSq[0]
    startCol = startSq[1]
    endRow = endSq[0]
    endCol = endSq[1]
    piece = board[startRow][startCol][1]
    pieceColor = board[startRow][startCol][0]
    if piece == 'P':
        if not movePawn(startRow, startCol, endRow, endCol, pieceColor):
            validFlag = False
    elif piece == 'R':
        if not moveRook(startRow, startCol, endRow, endCol):
            validFlag = False
    elif piece == 'N':
        if not moveKnight(startRow, startCol, endRow, endCol):
            validFlag = False
    elif piece == 'B':
        if not moveBishop(startRow, startCol, endRow, endCol):
            validFlag = False
    elif piece == 'Q':
        if not moveQueen(startRow, startCol, endRow, endCol):
            validFlag = False
    elif piece == 'K':
        if not moveKing(startRow, startCol, endRow, endCol):
            validFlag = False
    if validFlag:
        return True
    else:
        return False

```

def isValid(startSq, endSq) -

This function checks whether the move we want to make is valid or not. It takes two arguments, startSq and endSq. The startSq argument contains the row and column of the current tile of the piece that has been selected. And the endSq argument contains the row and column of the tile we want to move the piece.

So we get startCol and startRow from startSq and endCol and endRow from endSq. We also find out what the piece is that we want to move and the color of the piece whether it is black or white. Then depending on the piece we call the functions accordingly which checks the validity of that move.

Here are the validity check functions for each of the chess pieces -

```
unknown
def movePawn(startRow, startCol, endRow, endCol, pieceColor):
    if pieceColor == 'b' and startRow == 1 and ((startRow < endRow <= startRow + 2) or (
            startRow < endRow and (endCol == startCol + 1 or endCol == startCol - 1))):
        return True
    elif pieceColor == 'b' and startRow != 1 and ((startRow < endRow == startRow + 1) or (
            startRow < endRow and (endCol == startCol + 1 or endCol == startCol - 1))):
        return True
    elif pieceColor == 'w' and startRow == 6 and ((startRow > endRow >= startRow - 2) or (
            startRow > endRow and (endCol == startCol + 1 or endCol == startCol - 1))):
        return True
    elif pieceColor == 'w' and startRow != 6 and ((startRow > endRow == startRow - 1) or (
            startRow > endRow and (endCol == startCol + 1 or endCol == startCol - 1))):
        return True

    return False
```

```
unknown
def moveRook(startRow, startCol, endRow, endCol):
    if startCol != endCol and startRow != endRow:
        return False
    else:
        return True
```

```
unknown
def moveKnight(startRow, startCol, endRow, endCol):
    if (endCol == startCol + 1 or endCol == startCol - 1) and (endRow == startRow - 2 or endRow == startRow + 2):
        return True
    elif (endCol == startCol + 2 or endCol == startCol - 2) and (endRow == startRow - 1 or endRow == startRow + 1):
        return True
    else:
        return False
```

```
unknown
def moveBishop(startRow, startCol, endRow, endCol):
    if endCol > startCol and endRow > startRow:
        colMoved = endCol - startCol
        rowMoved = endRow - startRow
        if colMoved == rowMoved:
            return True
    elif endCol < startCol and endRow > startRow:
        colMoved = startCol - endCol
        rowMoved = endRow - startRow
        if colMoved == rowMoved:
            return True
    elif endCol < startCol and endRow < startRow:
        colMoved = startCol - endCol
        rowMoved = startRow - endRow
        if colMoved == rowMoved:
            return True
    elif endCol > startCol and endRow < startRow:
        colMoved = endCol - startCol
        rowMoved = startRow - endRow
        if colMoved == rowMoved:
            return True
    else:
        return False
```

```
unknown
def moveQueen(startRow, startCol, endRow, endCol):
    if moveBishop(startRow, startCol, endRow, endCol) or moveRook(startRow, startCol, endRow, endCol):
        return True
    else:
        return False
```

```
unknown
def moveKing(startRow, startCol, endRow, endCol):
    if endRow == startRow + 1 or endRow == startRow - 1 or endCol == startCol + 1 or endCol == startCol - 1:
        return True
    else:
        return False
```

If the move of the piece is valid, then the **move(startSq, endSq)** function is called.

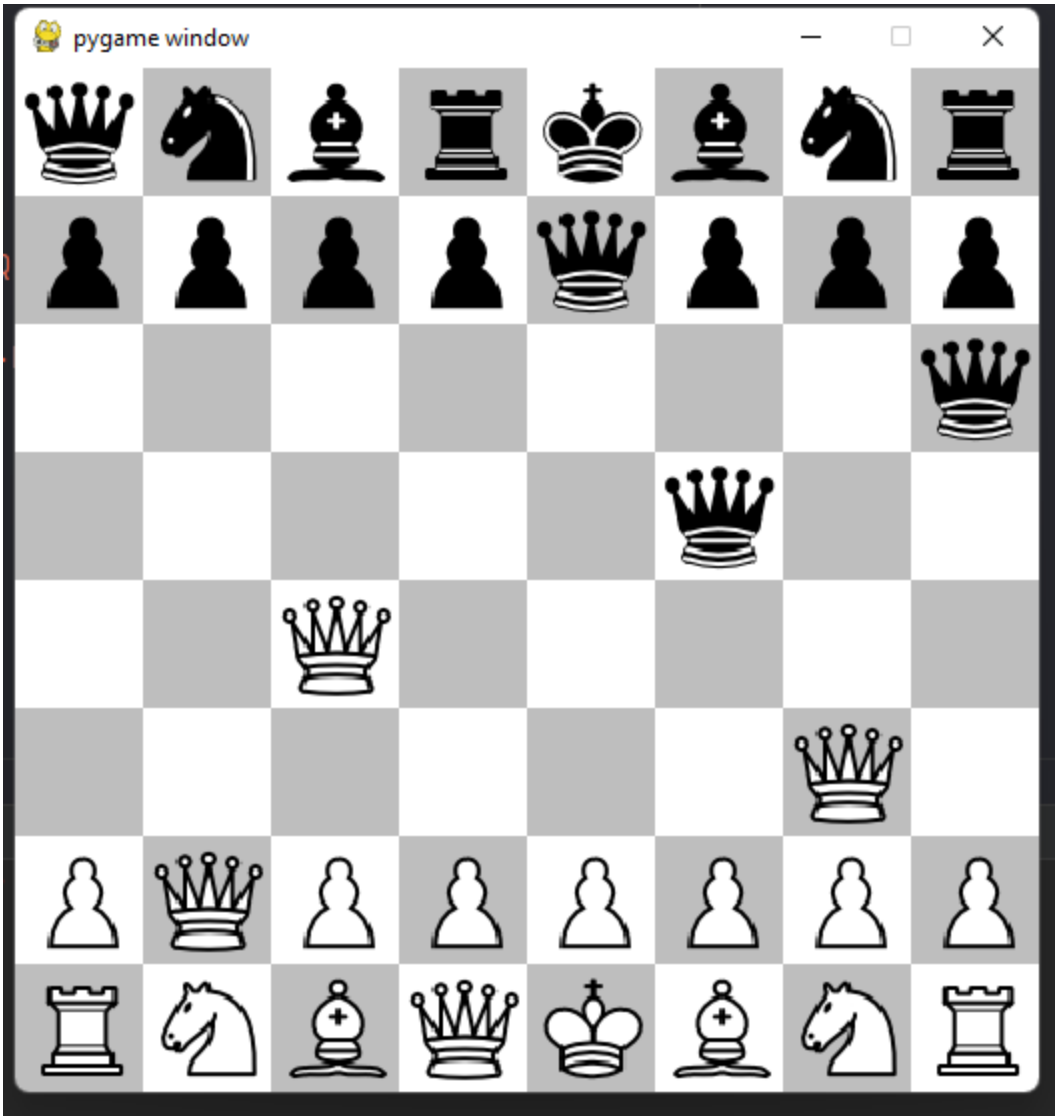
```
def move(startSq, endSq):
    startRow = startSq[0]
    startCol = startSq[1]
    endRow = endSq[0]
    endCol = endSq[1]
    pieceCaptured = board[endRow][endCol]
    pieceMoved = board[startRow][startCol]
    movedColor = board[startRow][startCol][0]
    print("movedColor:" + movedColor)
    capturedColor = board[endRow][endCol][0]
    print("capturedColor:" + capturedColor)
    if movedColor != '-' and capturedColor != '-' and startSq != endSq and movedColor == capturedColor:
        print("You cannot attack your own")
    else:
        board[startRow][startCol] = '--'
        board[endRow][endCol] = pieceMoved
        if pieceCaptured == 'wK':
            print("BLACK HAS WON")
        elif pieceCaptured == 'bK':
            print("WHITE HAS WON")

```

The move function also takes startSq and endSq as arguments and calculates startRow, startCol, endRow and endCol from it. It also finds out the tile we want to go and the piece we want to move. Also finds out the color of the piece we want to move and if we want to capture a piece, it also finds out the color of that piece. And if two colors are the same it prints out a message that "You cannot attack your own pieces". But if that is not the case, it moves the selected piece to the selected tile. If the selected tile had a king of a certain side then that side lost, and the program prints out which side has won.

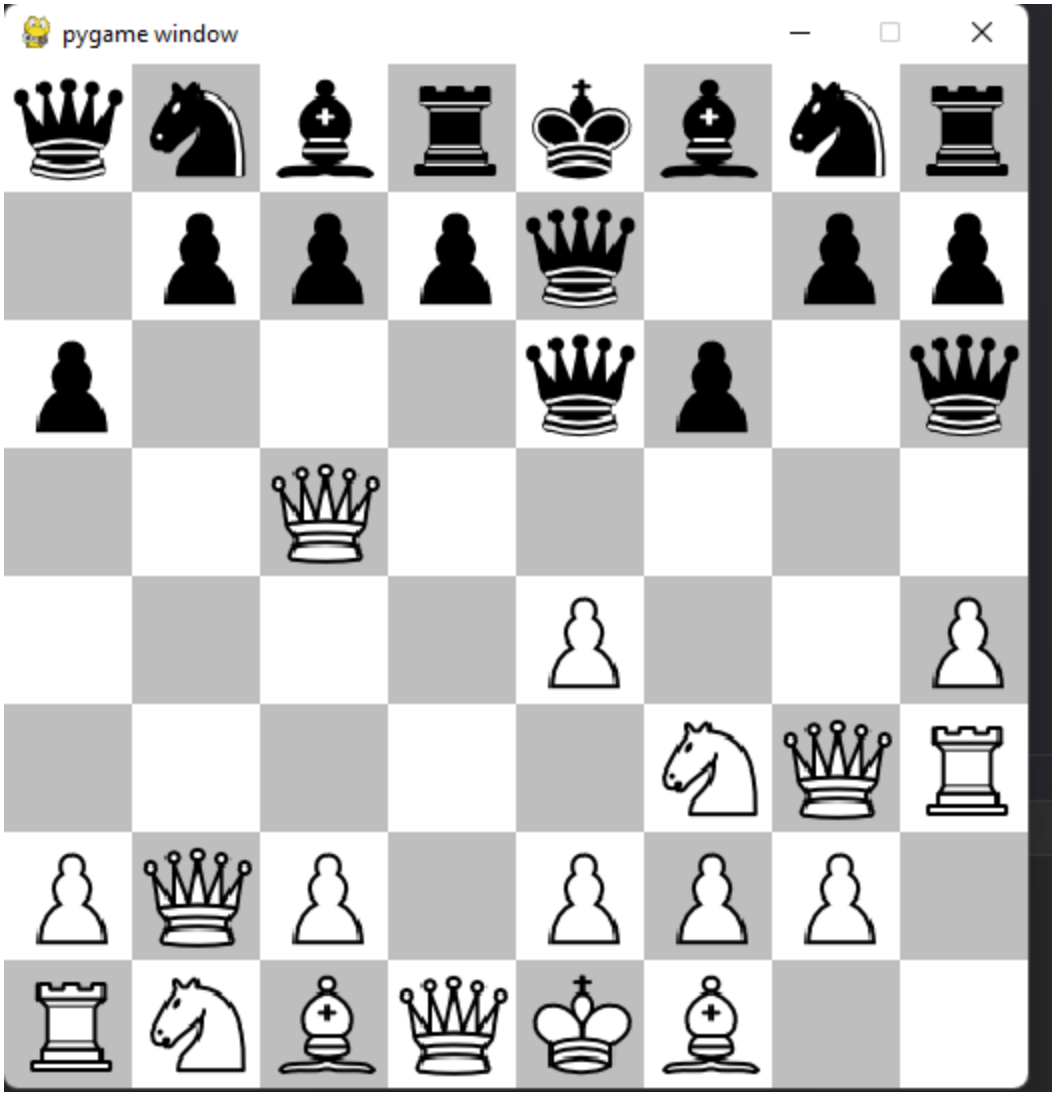
Gameplay Cases:

Now we will see how our game handles our implemented cases.

- Initially the board looks like this -

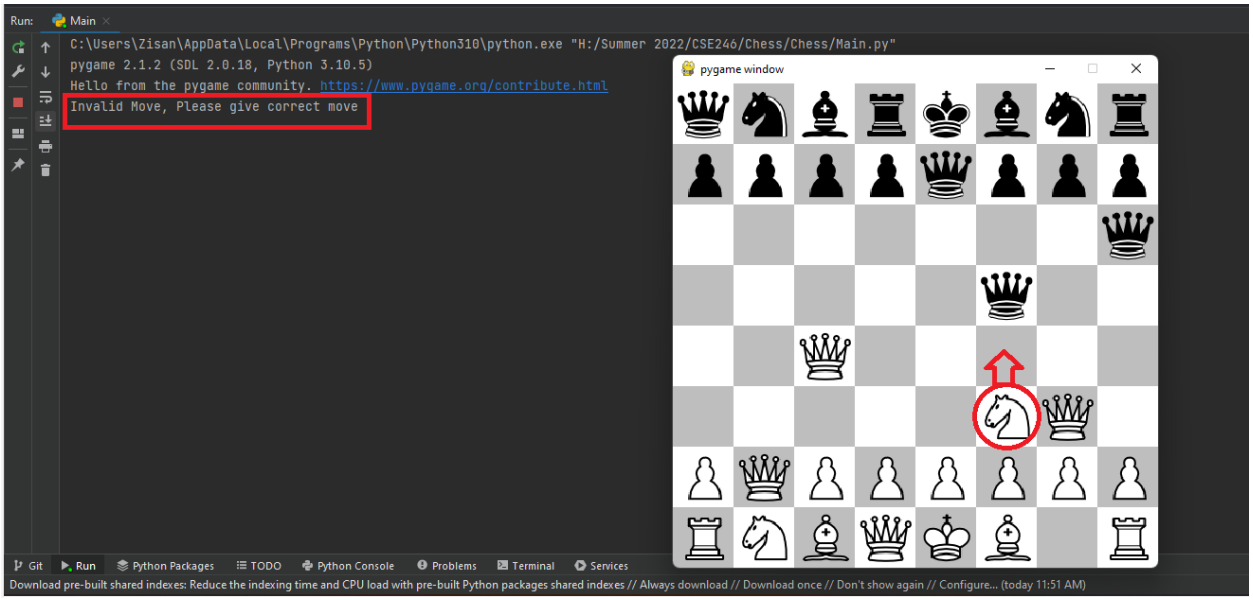


- Now if we want to move some chess pieces, we can do that by using mouse clicks.



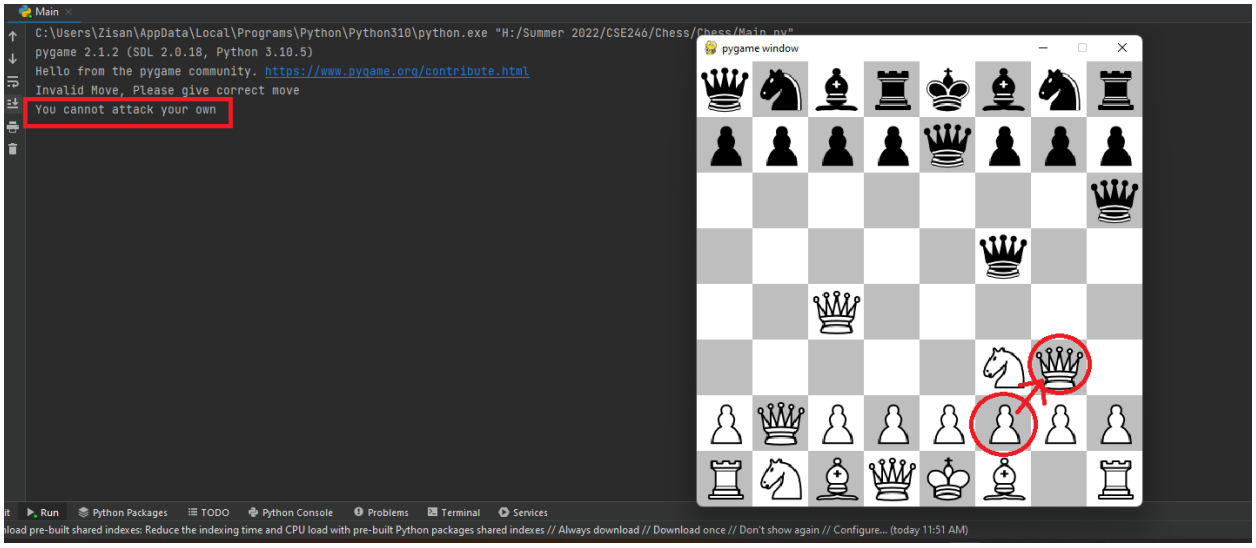
Here we have randomly moved some pieces.

- Now we will check out the incorrect move feature. We will incorrectly move the knight to show that the program warns us about the incorrect move. But this feature will work for any piece.



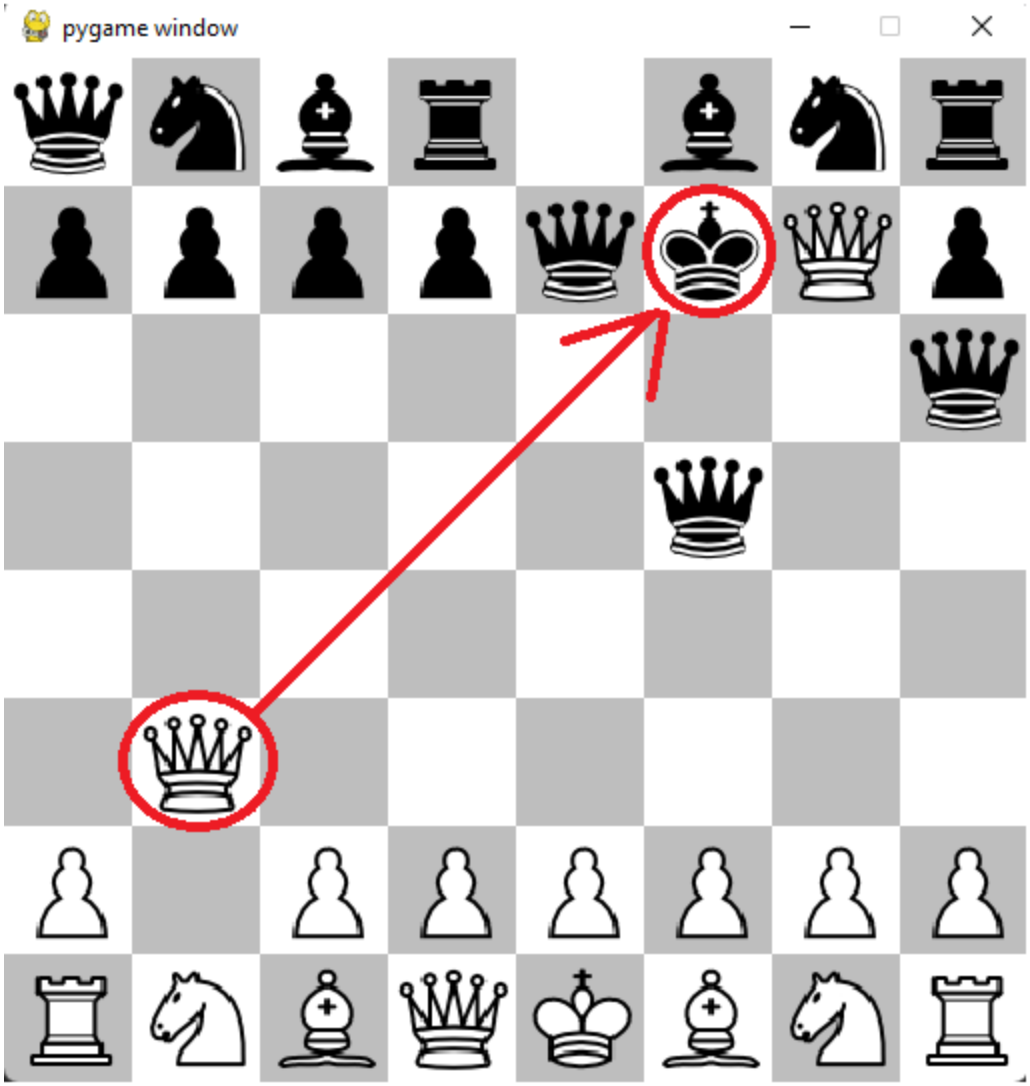
Here we wanted to move the knight one tile upward, which is not a valid move for the knight and the program warned me.

- Now wewe will check what the program does if wewe attack a piece of my own.

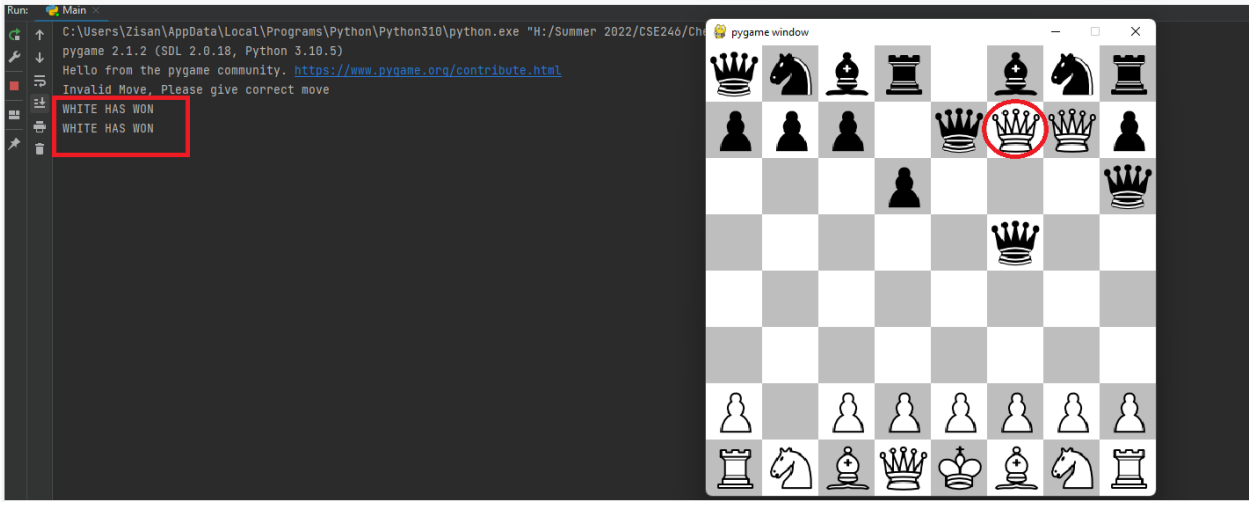


Here we wanted to attack the white queen with my white pawn. And the program gave us a warning.

- Now we will check the simple win or lose system of the game. Here you can see that we have created a checkmate situation.



Now if the white queen kills the black king, then white will win and the game shows it.



Same will happen if black wins.

Conclusion:

So this was our project. We tried to implement a two player chess game and also tried to implement the n-queens algorithm in the chess game. In the end, we were successful in making a simple two player chess game with n-queens implementation. In the project report we explained the code of the project and showed how our game handles the various gameplay cases. In this future, we would also like to implement the future prospects of our project.