CSE 274 Fall 2019
Project 06 – Hashing, Dictionary and Game Moves
Points: 100

Learning Outcomes:

- Define a class to represent a game board
- Generate game board configuration using recursion
- Determine the best move for a given game board configuration
- Store game moves in a dictionary
- Use your own defined class to work with a data structure (Dictionary, Hashing)

Naming requirements:

- Not following any of these naming requirements may result in a score of 0.
- You should have a class named **Board** in the default package. This class is where you store a Tic Tac Toe board as an array of 9 values.
- You should have a class named **TicTacToe** in the default package. This class is where the logic of the game will be. This is the class that will contain a Dictionary for storing Tic Tac Toe boards.
- You should have a class named **TicTacToeInteraction** in the default package. This is where the user interaction for the game should be. This is where the main() method should be.

Project Goal:

Consider all the possible boards in a game of Tic Tac Toe. For each board position, there is a "best move" (or multiple moves that are tied for best). 4th Edition Textbook: see programming project 6 in chapter 19 (page 573), 5th edition textbook: see programming project 7 in chapter 20 (page 587).

Write a program that plays a game of Tic Tac Toe in which you use a dictionary to calculate and store all possible board positions, and the best move associated with that position.

Specifics:

- Each board position should be stored as an array (one- or two-dimensional) of 9 values, in the class **Board**, representing who is occupying each of the 9 squares. Note that a square can be occupied by X or by O, or it can be empty.
- The board position should be the key for the dictionary. The type of array you choose to use is up to you. Choose something that is easy to work with. All you are trying to do is indicate whether each of the 9 squares is X, O, or empty.
- In our game, X will always make the first move.
- The value associated with each board should be the best move that can be made in that position. Again, the data type you choose to use is up to you. The move could be represented by a row and column, or by a number in the range 0 to 8. Note that you should not have to indicate WHO moves (it should be obvious from the board). It should only have to indicate where to move. If there is more than one best move, it is up to you which of those best moves you choose to return.
- **Important:** thus, the key-value pair in your dictionary will be a board (key) and a move (value).
- Although there are 3^9 board configurations (in which each square is X, O, or empty), many of those configurations are not valid in the game of Tic Tac Toe. For example, you cannot have a board in which X appears 7 times and O appears only 1 time. Similarly, you cannot have a board where both players have 3 in a row. You also cannot have a board where O appears 3 times and X appears 2 times (because X goes first). You only need to store valid board positions in your dictionary. It is OK if you want to store ALL positions in the dictionary, but then you should have a plan for what the "best move" value will be for each of those positions, even in the board is not valid. You can use recursion

(recommended) to generate all valid boards. You can also use an iterative process. Please take a look at the following figure to understand the process of making a valid move and generate valid boards.

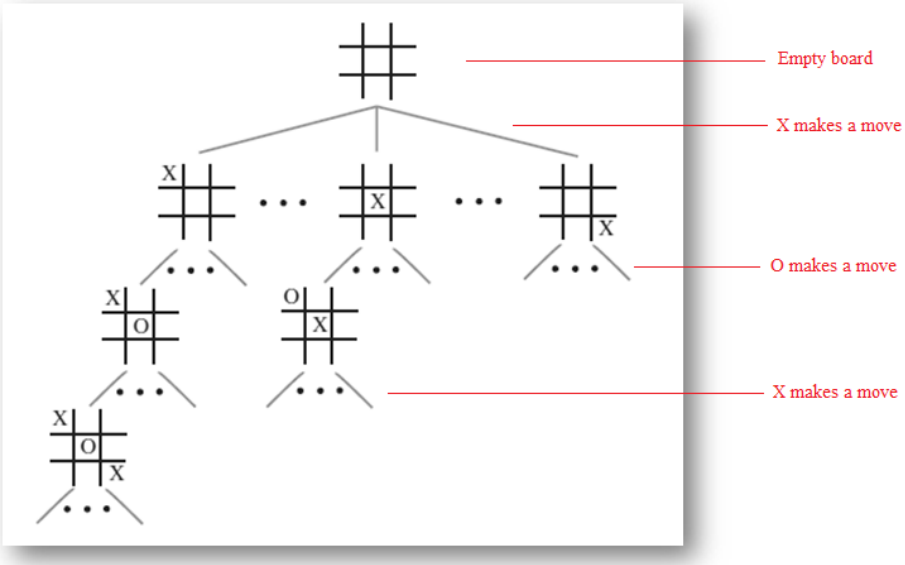


Figure: Step by step illustration of valid moves and associated boards starting with an empty board

- **Important:** We can write a recursive method to generate all valid boards based on the sequence of moves players make. Here is the pseudo code for the recursive method:

```
generateBoards(Board board, int numberOfEmptySpaces, boolean move){
        if numberOfEmptySpaces is 0:
                return. // No more empty spaces to make a move

        // We have to check all 9 spaces to find the empty spaces to make a move
        for loc = 0 to totalNumberOfSpaces - 1:
                if board[loc] is empty:
                        if move is true:      // we're assuming that X makes a move when it's true, O makes a move otherwise
                                board[loc] = 'X'
                        else:
                                board [loc] = 'O'
                        // Now we have a new board and must calculate the best move for this board
                        // You can also check to see whether it's a valid board or not before adding it to the game dictionary
                        gameDictionary.add (board, findBestMove(board))
                        generateBoards (board, numberOfEmptySpaces – 1, !move)
                        board[loc] = '-'      // Undo the move we made to the loc so that we can
}
```

You can call this recursive method with the initial board configuration where we have an empty board with nine empty spaces and X makes the first move. This can be done by the following:

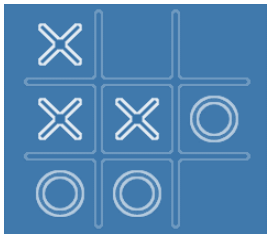
```
generateBoards(new Board("---------"), 9, true)
```

It will then call the recursive method and generate all valid boards just like the above figure.

- Best move: To determine the best move we can do the following in order:
  1. Can I win?
     Look at all the empty spaces to check whether making a move would end up winning the game.
  2. Is the other player winning?
  3. If none of the above two are true, then we can maximize our chance to win this game by checking two future moves.
     Make a move and then another, if there is an empty space, and then check whether that generates a winning board. If it does, then it's the best move.
- To store the board positions and associated best moves, you must use the given Dictionary implementation, HashedDictionary. The DictionaryInterface, a driver class and the Name class required for the driver class are also given. Check and run to be comfortable using these classes.
- To interact with the user, you may use either a console-based or GUI-based game. Only choose GUI if you are very comfortable working with Java GUIs. If you use a console-based game, allow the human to always go first and always show the board before each human move. The human should enter her move as a single digit number in the range 0 through 8, where 0, 1, 2 are the top three cells, 3, 4, 5 are the middle three, and 6, 7, 8 are the top three.
- You get to decide what methods and constructors you want for the **TicTacToe** class. However, there is one public method that the **TicTacToe** class must have, in order to help me with grading your work:

```
public int getBestMove(String board){
        . . . . . .
}
```

- This method should take a board in the form of a string and return the best move as an int in the range 0 through 8. This method should not need much code. It should really just rely on your Dictionary to retrieve the correct move for a given board (first using the string to create a Board object and use it as a key). Here is how this method should work. Suppose you have the following board:



It must be X's move. The best move is to place the X in the lower-right corner. If we number the squares 0 through 8 (by beginning at the top-left and going left-to-right, top-to-bottom), we would represent this board with the string "X--XXOOO-". Thus, getBestMove("X--XXOOO-") should return 8 indicating that the best move is the lower-right corner. Similarly, for the board below, getBestMove("--OXXO--X") should return 0, because the best move for O right now is in the upper-left corner, blocking X from winning.

Just to re-emphasize: the getBestMove() is essential for assisting your instructor in testing your code. It does not mean you must store boards in the dictionary as strings. You could save this method for last if you want, because you do not need it for other parts of the program to work.

Rubric:

Task	**Max score**
Board class: - Use an array to store a tic tac toe board, 5 points - Implements hashCode method, 15 points - Implements equals method, 5 points	25

TicTacToe class:
- Uses a dictionary to store boards and associated best moves, 10 points
- Generates and stores all valid boards in a dictionary, at most 30 points
  o Dictionary returns a move for some valid boards, but not for all valid boards: 10 points
  o Dictionary returns a move for all valid boards, but sometimes that move is not a legal move: 15 points
  o Dictionary returns a legal move for all valid boards, but sometimes that move is not a best move: 20 points
  o Dictionary returns a best move for all valid boards: 30 points
	40

TicTacToeInteraction: - Tic Tac Toe user interface works correctly, 15 points - Shows game results (who wins, draw, etc.) correctly, 10 points	25
getBestMove(String) implemented correctly, returning an int for a given valid board	10
Code formatted according to generally accepted standards. This includes Javadoc comments for the TicTacToe class.	0 (deductions only)