Name: Folawiyo Campbell
Course: EN 685.621
Email: fcampbe6@jh.edu

Best Move

The AI’s best moves are stored in a TreeMap implementation with the board priority values as keys and board coordinates as values. The AI’s board priority is of the form:

8	4	7
3	9	2
6	1	5

Whenever the AI needs to get the best move it simply polls the last entry in the TreeMap. The TreeMap is built on a red and black tree with log(n) time insertions and retrievals. The ``aiPlay()`` function shows how this is used.

Conditional Statement

This is implemented in the ``aiPlay()`` function.

Running time analysis

Here we will analyze the ``aiPlay()`` algorithm. This is the algorithm that is executed when the AI is called upon to play.
When the AI is called it does the below steps:

- Checks if it can win and makes that move. This takes constant time. The function executed for this check is ``winningPlace()``. Winning place delegates to ``winningDirection()`` which checks if the player (in this case the AI) has a place it can win. ``winningDirection()`` checks all rows first, then all columns, and then the two diagonals. The Tic-Tac-Toe board is always a board of 9 places so ``winningDirection()`` always checks 9 spots for horizontal, 9 spots for vertical, and 6 spots for diagonal winning possibilities. Since this never changes we can say ``winningPlace()`` does constant time work. If there is a winning spot it assigns its symbol to that spot which is also constant time. We can represent this with c1.
- If it can’t win it checks if it needs to block and makes that move. This also takes constant time as it does the same steps it needed to do to check if it can win but for its opponent. We can represent this with c2.
- If it does not have a winning play and it does not need to block it will retrieve an available best move. The best moves are stored in a TreeMap built on a red and black tree. A retrieval from this data structure will take log(n) time. We can approximate this as c3log(N).

In total, we can infer that the work for the AI play would be **Tn = c1 + c2 + c3log(N)** where N is the number of spots on the board. This gives us **O(logN)**.

I have uploaded the complete code on GitHub here and have also attached it to the assignment zip. I implemented functionality such that I can simulate a human and AI player in my main function here.

The output of my simulation can be found below:

```
Starting Board State
[[null, null, null], [null, null, null], [null, null, null]]
Human Player played
[[null, null, null], [null, O, null], [null, null, null]]
Ai Player played
[[X, null, null], [null, O, null], [null, null, null]]
Human Player played
[[X, O, null], [null, O, null], [null, null, null]]
Ai Player played
[[X, O, null], [null, O, null], [null, X, null]]
Human Player played
[[X, O, O], [null, O, null], [null, X, null]]
Ai Player played
[[X, O, O], [null, O, null], [X, X, null]]
Human Player played
[[X, O, O], [null, O, null], [X, X, O]]
Ai Player played
[[X, O, O], [X, O, null], [X, X, O]]
```

Simulation explanation:
The output shows a String explaining who played followed by the board state after the player played.

- First is the starting board state.
- The human player plays first and plays in the center of the board at row =1, column =1.
- Next is the AI's turn, it plays and picks its best available move which is the top left corner. The human already played the AI's very best move in the center.
- The human plays next at row =0 column =1
- The AI sees that the human is about to win and blocks at row=2 column =1.
- The human player plays at row=0 column 3
- The AI plays and blocks the human at row =2, column =1
- The human plays at row =2, column=2 to block the AI
- The AI wins in the available vertical winning direction playing at row =1, column =0