1. *# """\  
   # @ @  
   # / /  
   # @ - A - B  
   # \\ / \\  
   # @ - C @  
   # \\  
   # @"""*

**@ @  
 / /  
 @ - A - B @  
 / \\ / \\ /  
@ - C - D - E  
 \\ / \\ / \\  
 @ - F - G @  
 \\ \\  
 @ @"""**

**elif** self.length == 3:  
 self.board = **"""\  
 @ @  
 / /  
 @ - A - B @  
 / \\ / \\ /  
 @ - C - D - E @  
 / \\ / \\ / \\ /  
@ - F - G - H - I  
 \\ / \\ / \\ / \\  
 @ - J - K - L @  
 \\ \\ \\  
 @ @ @"""**

*@ - C - D - E @  
# / \\ / \\ / \\ /  
# @ - F - G - H - I @  
# / \\ / \\ / \\ / \\ /  
# @ - J - K - L - M - N  
# \\ / \\ / \\ / \\ / \\  
# @ - O - P - Q - R @  
# \\ \\ \\ \\  
# @ @ @ @"""*

*# """\  
# @ @  
# / /  
# @ - A - B @  
# / \\ / \\ /  
# @ - C - D - E @  
# / \\ / \\ / \\ /  
# @ - F - G - H - I @  
# / \\ / \\ / \\ / \\ /  
# @ - J - K - L - M - N @  
# / \\ / \\ / \\ / \\ / \\ /  
# @ - O - P - Q - R - S - T  
# \\ / \\ / \\ / \\ / \\ / \\  
# @ - U - V - W - X - Y @  
# \\ \\ \\ \\ \\  
# @ @ @ @ @"""*

**def** is\_over(self) -> bool:  
 *"""  
 Return whether or not this game is over.* **:return***: True if the game is over, False otherwise.* **:rtype***: bool  
  
 """  
 # Count the number of '@' innitially* status = **False  
 if** self.get\_current\_player\_name() == **'p1'**:  
 status = **True** n = 0  
 s = StonehengeState(status, self.length)  
 **for** li **in** s.info:  
 **for** ele **in** li:  
 **if** ele == **'@'**:  
 n += 1  
 *# Count the number of leylines claimed in current state.* n1 = 0  
 n2 = 0  
 i = 1  
 **for** ele **in** self.info[0]:  
 **if** ele == **'1'**:  
 n1 += 1  
 **elif** ele == **'2'**:  
 n2 += 1  
 **while** i != self.length + 2:  
 **if** self.info[i][0] == **'1'**:  
 n1 += 1  
 **if** self.info[i][0] == **'2'**:  
 n2 += 1  
 **if** self.info[i][-1] == **'1' and** i != self.length:  
 n1 += 1  
 **if** self.info[i][-1] == **'2' and** i != self.length:  
 n2 += 1  
 i += 1  
 **for** ele **in** self.info[-1]:  
 **if** ele == **'1'**:  
 n1 += 1  
 **elif** ele == **'2'**:  
 n2 += 1  
 **if** n1 >= 1/2 \* n **or** n2 >= 1/2 \* n:  
 **return True  
 return False**

status = **False  
if** self.current\_state.get\_current\_player\_name() == **'p1'**:  
 status = **True**n = 0  
s = StonehengeState(status, self.current\_state.length)  
**for** li **in** s.info:  
 **for** ele **in** li:  
 **if** ele == **'@'**:  
 n += 1  
*# Count the number of leylines claimed in current state.*n1 = 0  
n2 = 0  
i = 1  
**for** ele **in** self.current\_state.info[0]:  
 **if** ele == **'1'**:  
 n1 += 1  
 **elif** ele == **'2'**:  
 n2 += 1  
**while** i != self.current\_state.length + 2:  
 **if** self.current\_state.info[i][0] == **'1'**:  
 n1 += 1  
 **if** self.current\_state.info[i][0] == **'2'**:  
 n2 += 1  
 **if** self.current\_state.info[i][-1] == **'1' and** i != self.current\_state.length:  
 n1 += 1  
 **if** self.current\_state.info[i][-1] == **'2' and** i != self.current\_state.length:  
 n2 += 1  
 i += 1  
**for** ele **in** self.current\_state.info[-1]:  
 **if** ele == **'1'**:  
 n1 += 1  
 **elif** ele == **'2'**:  
 n2 += 1  
**if** n1 >= 1 / 2 \* n **or** n2 >= 1 / 2 \* n:  
 **return True  
return False**

**def** recursive\_minimax\_score(game: Game, state: GameState, player: str) -> int:  
 *"""  
 Return the score for each possible move.  
 """* scores = {}  
 **if** game.is\_over(state):  
 **return** 1 \* (check\_over\_score(game, state, player))  
 **else**:  
 moves = state.get\_possible\_moves()  
 *# return max([recursive\_minimax\_score(game, move) for move in moves])* **for** move **in** moves:  
 next\_state = state.make\_move(move)  
 scores[next\_state] = (recursive\_minimax\_score(game, next\_state, player))  
 *# scores.append(recursive\_minimax\_score(game, next\_state, player))  
 # print(recursive\_minimax\_score(game, next\_state))* **return** max(scores)

**def** recursive\_minimax\_score(game: Game, state: GameState, player: str) -> int:  
 *"""  
 Return the score for each possible move.  
 """* scores = []  
 **if** game.is\_over(state):  
 **return** 1 \* (check\_over\_score(game, state, player))  
 **else**:  
 moves = state.get\_possible\_moves()  
 *# return max([recursive\_minimax\_score(game, state.make\_move(move), player) for move in moves])* **for** move **in** moves:  
 next\_state = state.make\_move(move)  
 scores.append(recursive\_minimax\_score(game, next\_state, player))  
 *# print(recursive\_minimax\_score(game, next\_state))  
 # for score in scores:  
 # print(score)* **return** max(scores)

scores = {}  
**if** game.is\_over(state):  
 **return** 1 \* (check\_over\_score(game, state, player))  
**else**:  
 moves = state.get\_possible\_moves()  
 *# return max([recursive\_minimax\_score(game, move) for move in moves])* **for** move **in** moves:  
 next\_state = state.make\_move(move)  
 score = recursive\_minimax\_score(game, next\_state, player)  
 scores[next\_state.\_\_str\_\_()] = score  
 scores[state.\_\_str\_\_()] = score  
 **return** score

scores = {}  
**if** game.is\_over(state):  
 **return** 1 \* (check\_over\_score(game, state, player))  
**else**:  
 moves = state.get\_possible\_moves()  
 *# return max([recursive\_minimax\_score(game, move) for move in moves])* **for** move **in** moves:  
 next\_state = state.make\_move(move)  
 score = recursive\_minimax\_score(game, next\_state, player)  
 scores[next\_state.\_\_str\_\_()] = score  
 **if** score == 1 **or** score == 0:  
 scores[state.\_\_str\_\_()] = score  
 **return** score  
 scores[state.\_\_str\_\_()] = -1  
 **return** -1

scores = {}  
**if** state.\_\_str\_\_() **in** scores:  
 **return** scores[state.\_\_str\_\_()]  
**if** game.is\_over(state):  
 **return** 1 \* (check\_over\_score(game, state, player))  
**else**:  
 moves = state.get\_possible\_moves()  
 **for** move **in** moves:  
 next\_state = state.make\_move(move)  
 score = recursive\_minimax\_score(game, next\_state, player)  
 scores[next\_state.\_\_str\_\_()] = score  
 **if** score != -1:  
 scores[state.\_\_str\_\_()] = score  
 **return** score  
 scores[state.\_\_str\_\_()] = -1  
 **return** -1