**Note – major changes made since original version of AI are highlighted in yellow, and bolded names in brackets are the names of the methods associated with each step or group of steps**

**Final Reversi Insane AI Algorithm : Pseudo-Pseudocode**

* The basic data such as piece positions, score, and turn from ReversiBoard is copied into LogicBoard, which is a bare-bones version of ReversiBoard used by the AI for its calculations
* LogicBoard is put through mini-max to get best move for the AI **(prettyDamnUnbeatable – where the data is copied into the LogicBoard and mini-max is called on that LogicBoard to return the best move)** 
  + Mini-max searches deeply in the beginning and late-middle game since the list of possibilities for the outcomes are low to begin with and get lower as the empty squares are filled up
  + Mini-max searches VERY deeply near the end-game since the list of possibilities for the outcomes become VERY small as the game board fills out
  + Calling mini-max on the LogicBoard does the following:
    - Initializing the mini-max **(zeroDepthHeuristicMaximize – which seeks to maximize board value for the AI while returning a co-ordinate to move to at the end of the recursion)** 
      * Generates and ArrayList of all possible outcomes (boards) of AI’s moves
      * Generates all possible AI moves
      * Prunes all the possible outcomes unless the board is near end-game using a **pruning method**
      * Goes through all possible outcomes in a for each loop
      * Calls minimize on each of the possible LogicBoard outcome to get a board value
      * Looks for the highest board value from the AI’s perspective
      * It tracks the index of the highest board value, which will be used to get the associated move, this is because the possible AI moves and possible outcomes of AI moves are related ArrayLists
      * It returns the best move using the best index it determined during the loop
    - During the minimize **(heuristicMinimize - which seeks to minimize board value for the AI)**:
      * If maximum depth or end-game (no moves left) is reached, it first modifies multipliers depending on piece count using a **multiplier modifier method**, then returns the board value from the AI’s perspective
      * Generates and ArrayList of all possible outcomes of human player’s moves
      * Prunes all the possible outcomes unless the board is near end-game using a **pruning method**
      * Goes through all possible outcomes in a for each loop
      * Calls maximize on each of the possible outcomes to get a board value
      * Looks for the lowest board value from the AI’s perspective
      * After going through the for each loop, it returns the lowest value found so far
    - During the regular maximize **(heuristicMaximize – which seeks to maximize board value for the AI)** 
      * If maximum depth or end-game (no moves left) is reached, it first modifies multipliers depending on piece count using a **multiplier modifier method**, then returns the board value from the AI’s perspective using a **board evaluation method**.
      * Generates and ArrayList of all possible outcomes of AI’s moves
      * Prunes all the possible outcomes unless the board is near end-game using a **pruning method**
      * Goes through all possible outcomes in a for each loop
      * Calls minimize on each of the possible outcomes to get a board value
      * Looks for the highest board value from the AI’s perspective
      * After going through the for each loop, it returns the highest value found so far

**Methods and Variables Related to the INSANE AI**

* Board Evaluation (**evaluateBoard – which evaluates the board based on four parameters**)
  + Always done from the AI’s perspective regardless of whose turn it is
  + Looks at four things when evaluating a board: stability, mobility, score, and position
  + Stability **(getStableDiscValue – finds the value of stable discs for a player)** 
    - Stability can only be calculated for very obvious cases
    - Uses a method outside of the evaluateBoard method
    - Looks at how many pieces are un-flippable
      * Goes through every square
      * Looks for a piece
      * Checks in all 8 directions to see if they are filled
      * If they are filled the piece is stable
      * Gives corners and edges more weighing indirectly by searching individual corners and all squares found in straight lines extending from these corners – they are effectively counted twice.
  + Mobility **(noOfMovesPossible – finds number of moves possible for a player)**
    - Uses a method outside of the evaluateBoard method
    - Looks at how many moves a player can make
  + Score
    - Looks at the player’s score
    - Uses the instance variables of the instance of class
  + Position **(uses a static 2D array called heuristicsBoard – which values different parts of the board)** 
    - A class-wide heuristics board is used, the same size as the LogicBoard
    - Each square has a value assigned to it, corresponding to each square in the LogicBoard
    - The higher the value, the more important, the lower, the less
      * Goes through every square in the LogicBoard incrementing the values
    - Calculates values as a percentage of differences, for example – A and B are the mobility values for the AI and human player, instead of returning just A or (A + B), it now returns (100.0) ((A - B) / (A + B)).
    - Uses multipliers to give weight to different values, since the importance of values are not the same throughout the game
    - Multiplies all values by a multiplier, and adds all the results
    - Returns the combined values
  + In general, the higher the value, the bigger the difference between the AI and the player – in which the board is in the AI’s favour, and the lower, the bigger the difference between the AI and the player – in which the board is in the player’s favour.
* Multipliers
  + Used to give weighing to different heuristic values in Board Evaluation
* Multiplier Modification **(modifyMultipliers – changes multipliers depending on piece count)**
  + Multipliers are modified depending on piece count to reflect the changing importance of heuristic values
* Pruning **(cornerPrune and beginningCornerPrune)**
  + If any moves are found that take the corners, it modifies the list of possible outcomes to have only these moves
  + In the absence of corner-taking moves, it modifies the list of possible outcomes to have only moves that deny the enemy the ability to take corners
  + In the absence of corner-taking or corner-denying moves, it leaves the list of possible outcomes untouched, since it doesn’t matter at this point
  + This is because of the importance of corners in this game, and also serves as a pseudo-alpha-beta pruning mid-game, helping to cut down on the number of possibilities mini-max has to search through
  + In the mini-max initializer, it not only modifies the possible outcomes ArrayList, but also the possible moves ArrayList, this is because we want these two ArrayLists to stay parallel/related.
  + Pruning is only done until near the end-game, at which point corners no longer matter as much, only maximizing the score, a piece count check will be used to determine when multipliers will be modified
    - This allows the AI to search for game-deciding moves it may have overlooked when pruning was still done even in end-game.

**Reversi Hard AI (prettyUnbeatable - Made After Insane AI was Complete)**

* Same as the Insane AI, except it does not adjust for depth, and maintains a constant depth of search of 3 for the entire game, this gives the human player a great deal more “breathing room” against the AI in the end-game. This AI somewhat closely mimics a human player in terms of how far ahead it can see in a game.

**Reversi Normal AI (somewhatUnbeatable - Made After Insane AI was Complete)**

* Generates all possible outcomes for AI given a current board
* Looks for the move that allows it to take a corner, and returns these co-ordinates
* If no move can take the corner, it simply returns the move that results in the board with the best board value

**Reversi Easy AI (stupid - Made After Insane AI was Complete)**

* Generates all possible moves for AI given a current board
* Generates a random index
* Returns the co-ordinate element located at that index in the list of all possible moves

**Takeaways from Testing and Changes for Insane AI**

**Problems Later Found in Initial Reversi AI Algorithm by Looking Over the Code and Testing (Me AndKeyhan)**

* The AI was simply bad, we could beat it by random moves, or just by avoiding the corners until the end-game, where they would take it and flip the entire board.
* Some values were inherently far too high, such as positioning, taking a corner would increase a board value by hundreds easily, whereas others would only increase a board value by a dozen at most
* Stability did not value corners or edges highly enough, despite their importance, it only counted the number of stable pieces
* Differences were not used in the board evaluation, this was a problem because
  + The AI only sought to increase its board value, but not to maximize the lead it had on its opponent
  + This meant sometimes, when given the choice to increase its board value significantly, or increase its board value somewhat, but decrease the enemy’s board value (indirectly), it would only choose to increase its own board value significantly.
* Corners were not given priority by the AI, it would frequently lose corners or make moves that allowed the enemy to take corners

**Changes (First Round)**

* Multipliers
  + Used to give weighing to different heuristic values in Board Evaluation
* Board Evaluation
  + Instead of evaluating a board from the AI’s perspective, it would evaluate the differences of board values between the AI and the player, seeking to maximize the AI’s lead over the human player.
  + Instead of calculating raw heuristic values, it would calculate values as a percentage of differences, for example – A and B are the mobility values for the AI and human player, instead of returning just A or (A + B), it now returns (100.0) ((A - B) / (A + B)).
  + Now uses multipliers to give weight to different values, since the importance of values are not the same.
  + Stability Evaluation
    - Goes through entire board for normal stable pieces at first (surrounded in all 8 directions), but gives corners and edges more weighing indirectly by searching individual corners and all squares found in straight lines extending from these corners – they are effectively counted twice.
* Pruning
  + If any moves are found that take the corners, it modifies the list of possible outcomes to have only these moves
  + In the absence of corner-taking moves, it modifies the list of possible outcomes to have only moves that deny the enemy the ability to take corners
  + In the absence of corner-taking or corner-denying moves, it leaves the list of possible outcomes untouched, since it doesn’t matter at this point
  + This is because of the importance of corners in this game, and also serves as a pseudo-alpha-beta pruning mid-game, helping to cut down on the number of possibilities mini-max has to search through
  + This used in the mini-max
  + In the mini-max initializer, it not only modifies the possible outcomes ArrayList, but also the possible moves ArrayList, this is because we want these two ArrayLists to stay parallel/related.

**Problems Discovered by Testers (Others)**

* AI expanded too fast in the beginning and middle of the game, allowing the player to win by flipping the pieces en-masse in the end, when the AI had no moves left and constantly skipped its own turn
* AI seemed to prioritize the wrong things, or things that it shouldn’t have from the tester’s point of view, taking the corners when it could’ve taken a winning move for example.
* AI seemed to delay taking corners for too long, allowing game to be won by player by avoiding corners at first, and taking the corners as soon as the board was filled to flip everything and win.

**Changes (Second Round)**

* Multipliers
  + Multipliers are modified depending on piece count to reflect the changing importance of heuristic values
  + This allows the AI to prioritize things it should in different stages of the game, in the beginning for example, it will no longer over-expand, but seek to keep its options open by maximizing the possible number of moves it can make – and in the end-game, it will no longer care about anything other than score, since the end-game is when victory is decided, etc.
* Pruning
  + Pruning is only done until near the end-game, at which point corners no longer matter as much, only maximizing the score, a piece count check will be used to determine when multipliers will be modified
  + This allows the AI to search for game-deciding moves it may have overlooked when pruning was still done even in end-game.
* Depth of Mini-max
  + Depth of mini-max will vary depending on piece count.
    - It searches deeply in the beginning and late-middle game since the list of possibilities for the outcomes are low to begin with and get lower as the empty squares are filled up
    - It searches VERY deeply near the end-game since the list of possibilities for the outcomes become VERY small as the game board fills out

**Final Testing**

* Not a single person won against the AI out of 5 games each
* AI is performing satisfactorily now
* Only problem is that if someone can recall all the moves they made to a winning game, they can win against the AI by performing the same moves over and over, but this shouldn’t be too much of an issue, no one would possibly be neurotic enough to do something like that.