**Connect N**

ConnectN is a multiplayer game that allows interactive or non-interactive users to play connect 4 with a variable number of consecutive tiles to win. The interface prompts users to enter the # of tiles necessary to be consecutive to win the game. The user can play against another player or versus the computer. If they play against the computer, there is an option for an AI SmartPlayer function that utilizes complex algorithms to calculate the perfect move in that situation.

**How to Play**

The interface will direct you through the directions of how to play the game and will prompt the user for how many columns they wish their scaffold to hold. Then it will ask the user what column number they wish to drop their token within, if the column is not a possible value, it will inform the user and prompt them again for a column number. It will then print a visual of the game and which columns have tiles placed in them and the color of those tiles. After someone wins the game, the computer will inform the user and state who has won the game.

**Design of My Classes**

**Game:**

Play function : My game uses a play function that checks if the game is completed and then checks if the player whos turn it is is an interactive player or not. If it is an interactive player it will take the turn and then immediately display the screen. Iti s not interactive and it notices that it has been five turns then it will prompt the user to press enter to continue.

Completed function: My completed function goes through each column and starting at the lowest level of that column will find out what color is at that point and then call each of my functions (findHorizontal, findVertical, findDiagonalRight, findDiagonalLeft) to determine whether a certain number of that color are in a row in each direction. Each of those functions are created outside of the completed function, and use recursion. Essentially, for each of those functions if it finds out if the next step in that direction is that same color, then it recursively calls itself and checks if the next one is the same color as well. It keeps a tally and returns how many in a row in each direction. If that many in a row is equal to or greater than the amount needed to win then that color is set to winner and the completed function returns true. Otherwise returns false.

My taketurn function finds out if it is completed and if it is then it returns false. Otherwise it chooses a move, makes that move and then displays. It the increments a passed by reference variable that signifies who’s turn it is.

**Scaffold:**

My Scaffold class stores all my past moves in a stack of ints and my actual scaffold is a vector of a vector of ints. Each vector represents a column and the ints are the number of levels. For the amount of levels there are I push back onto my vector. For my checkerat function I pass in the actual names of the columns and levels (1,2,3….) and then fsubtract by one to figure out the actual value of that spot. For my make move I would make sure it was valid and not vacant and then push it onto the vector. FOr undomove I had used a stack so I just looked at the top value and then popped it and undid that move.

**Player:**

For my badplayer function I would just choose the left-most vacant move possible.

**SMARTPLAYER CHOOSE MOVE:**

My rating function -> so my rating function takes in a pointer to a scaffold (it always get’s passed in my copy of original scaffold though). It takes the color passed to it and makes it the computer color and then passes it to the completed function which will change the color to whicheveer color has won. It checks if the scaffold is not completed (nobody has won) and returns a 5 if there are still moves to be played.

If the computer has won it will return a large number plus the numberempty. This way we can make sure we chose the combination with the most number empty because that will be the quickest win. If it’s a tie game we return 0 and if its a human win we return -100 minus the numberEmpty to make sure we once again can tell which is the quickest win.

My determine Best computer move function is a pair of ints and one of the int’s contains the rating and the other int in the pair contains the column option that gives that rating. I create a copy of scaffold that I’ll pass so that I can makeMove’s without altering the real scaffold. I created a vector that stores pairs of int’s as my possible moves so I can keep track of the rating of each column move possible.

I cycled through each column and checked if the top of each level had a vacant because that means there was at least one possible move in that column. I would make the move and then get the rating for that move to find out if it was an immediate win or not. I then switched the color so that when I called the move again it would look for best move for the opposite player. I thIf the curr\_rating returned a 5 then I knew there was no winner and recursively called the determineBestComputerMove but with a new color each time and increased the depth by 1. When I finally got a winner I would push back that move onto my posMoves vector so I could later compare the ratings of each of the possible solutions I got. Then after I had stored the value I would undoMove and then iterate through the other colors. When it finally broke out of the for loop for the amount of columns, I would sent the collections of moves to a maxRating function if it was a computerwinner or the minrating function if it was a human player.

My minRating and maxrating helper functions take in all my posMoves and takes an iterator and compares the rating between each move in the vector. It will then return the value with either the highest or the lowest rating as well as the column that must be taken.

**2. PSUEDOCODES:**

**Completed function**

Go through each column

For each level at that column check what color

If the color was vacant

Move to the next level

If there was a marker at that spot

Go through and if upwards in the vertical, horizontal, diagonal right or diagonal left how many more markers in that color

If that number is greater than the number need

Return true and mark that color as the winner.

**FIND DIAGONAL RIGHT : (same format as find horizontal,vertical, etc…)**

Test if there are more columns to the right so that no undefined behavior

Return 1 if it’s the end because then the diagonal was size 1

If the color is not vacant at the location one above and one to the right

Recursively call the function while using that new location as the starting Level and add one

Else

Return one which will represent the size of the diagonal.

**TAKE TURN:**

If the game has a winner or a tie then return false immediately

Else

Check who’s turn it is by whether tracker is even or not

Make the move and display

Increment who’s turn it is by one and return true

**PLAY:**

While the game is not completed

If the player whose turn it is, is interactive -> take turn

Else

Taketurn but check if it has been 5 moves

If it has been five moves, prompt the user to press enter to continue

When the game is completed

Display the screen adn announce winner.

**determineBestComputerMove:**

Create a vector with to store all possible moves

Create a copy of staffold to add moves on to

If the scaffold is full

Return values that show no possible column is possible.

Repeatedly:

Go through each column

If the column is not full -> drop a checker in that column

Check if that checker leads to a win.

If it leads to a win record in the vector

If it does not lead to a win, see if the next player will win with their next move.

Continue until someone wins.

Record who wins as well as how many moves it took them.

Undo the latest move and check each possible win that starts with that first move.

Continue recording until all the moves that stem from that first move have go on.

Then iterate to attempt to start with the next column and continue this process.

Eventually break from the loop and then depending on the color find the column that leads to the max rating by calling a function that iterates through your vector of possible moves.

**Scaffold: makeMove:**

Take in Column names (1,2,3) as parameter to simplifying

Check that parameter is valid

If there is no vacant spot in that column, return false

Iterate to the next column

Else

Go through each level and insert at the lowest vacant spot.

Push the move on the pastMoves stack to account for the move

**ISSUES:**

I had a lot of issues but one that I did not realize until later was my completed function. I used recursion to check if there were patterns in my scaffold (findDiagonal, findHorizontal, etc…) and I did not realize this slowed down my program dramatically. I also struggled with reaching a lot of bad access spots when trying to input values into my columns. This came because I was trying to compare the column names with my spots on the scaffold. Another big issue that i had was that sometimes my function would not notice when I had diagonal patterns. I spent a while confused on this and then recognized that i was only checking a single direction. Additionally, I had issues where I would drop a checker and it would be above the pattern length and not register as a winner so I had to rework it so that it recognized those as wins.

SmartPlayer Issues:

I was unable to solve my issues with smartPlayer the biggest one being that when I would push my moves on to posMoves it would fail to keep all of them and at a given time by posMoves would only contain 1 to 2 moves at max. This made it impossible to compare the possible moves I was going to make and return the answer with the best rating. In addition, I really struggled to figure out how I would keep track of what column corresponded to the ratings until one of the LA’s suggested that I made my function return a vector of pairs so that I could account for both. My smartplayer recognizes what will result in an immediate loss but had trouble storing that information when I iterated to check what would happen in new columns.