Judy Wu
101184803

Connect 4 Game Project Report

---

1. Describe the problem(s) you solved and provide a brief description of the main pieces of your solution (general approach, data, algorithms).

TLDR; I used a lot of lists (list of list) to represent a board and a lot of for loops to loop through those two lists (checking for win or tie states), since it makes it easy for me to simulate a game of connect four, and while loops to validate input and a while loop for players to play against each other until the game is over (win or tie). A lot of lists were used, for validation functions, and even as 'nodes' potential boards for the minimax AI.

I used a lot of integers since I didn't have to run precise calculations (which you would use float) since they were just used to represent player discs. I also used recursive functions for minimax.

Unfortunately I didn't find much use in sorting or dictionaries, since I didn't need to sort the board, or anything like that and I didn't feel it was necessary to cache things either.

2. Describe how you applied computational thinking in solving the problem.

GENERAL PROBLEM SOLVING TO CREATE PRODUCT/IN DEPTH EXPLANATION

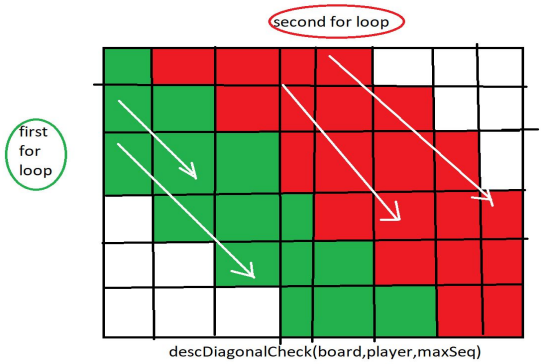
- Creating board - initboard()
    - The first thing I had to do was to create a representation of the board that I could parse through logically. I knew that the easiest way to do so was to use lists. This way I could represent rows and columns of the board. It would also make it easy for me to look into counting consecutive sequences, and adding discs since I could track where each player's discs would be.
        - So I created a function to create and initialize a 6 row by 7 column board. I ended up deciding to create a function since I could eventually keep score count, which would be simple/extra.
- Outputting the board - printboard(board)
    - Emoji trouble: Since it was originally meant for discord, I was going to print emojis so i needed to translate my 0s, 1s, and 2s into something like this :white_circle: (I just parsed through the board using 2 for loops and converted numbers into their respective emoji), or just a plain emoji since I was tired of looking at numbers and looking at colors was easier for me to spot

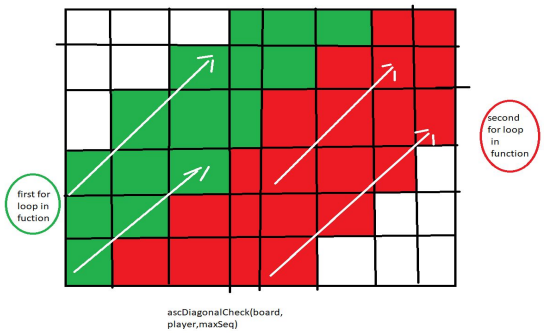
```
# import emoji module
import emoji


print(emoji.demojize("😃"))
print(emoji.demojize("😜"))
print(emoji.demojize("🤐"))
```

```
:grinning_face_with_big_eyes:
:winking_face_with_tongue:
:zipper-mouth_face:
```

patterns and bugs when bug testing (you can see the version I linked that prints out emojis instead on replit (written by me-wujudy, which is my lastname and firstname)).
    - I ended up leaving the emoji output out because I was struggling very much trying to look at discord python documentation (which was very very different from just coding regular code?) and using commands
        - I just used 0,1,2 to represent discs instead in console
    - I also knew that if you were to run the code in your console, the emojis would not properly output unless you were to install something that helps you out with it in your console and I didn't feel that it was appropriate to request that you did, so I just linked the version where it is the exact same code except it prints out emojis because that compiler can output it properly
- Adding a disc - addDisc(board, userInput, playerNum)
    - (I had to place it at the bottom-most open position and validate)
    - Validation:
        - In order to add a disc I didn't want it to add a disc and negate the player's move if they accidentally place it into a column that's already full, so every time before I accepted a users input I checked before with a function that checks if the column at the user's input was full or between 1 to 7.
    - Adding the disc: Since connect 4 is played vertically I had to 'simulate gravity' since it drops the disc instead of placing them at a row, to do this I just used a while loop to iterate over the column from the bottom and place it once it sees an empty spot.
- Input validation (double checking the user's input)
    - I just used conditionals to check if the input was less than 1 or greater than 7 along with if the column was full.
    - I checked the validity by using another function that iterated through the rows with the user's input as the column (I did the same for my AI input decisions as well)
        - If the counter counted that the row was full and equal to 6 elements then the row was full
        - If the row was full then I would tell the user their input is invalid since the row was full so they don't make the mistake again
- Checking State of the Game/simulating back and forth between players - checkWin()
    - I needed to do this since I had to while loop back and forth between the players move until the game was either a tie or someone won, since if the state was either tied or a player had won, the while loop should stop
    - I had to check if the game was over whenever a player played their move.

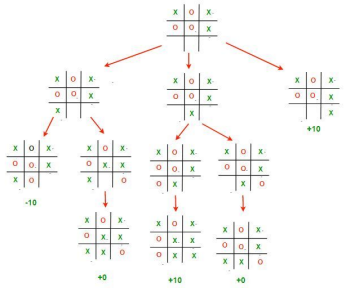
- There are 3 states of a game: someone won, there's a tie, or it is still in play session.
    - To check if someone won you had to check for horizontal consecutive player pieces of length 4 and for vertical wins you had to check for consecutive player pieces of length 4 as well. This was simple enough just by looking through all the rows and checking for consecutive player sequences for columns and rows. However for diagonal sequences I had to think a little harder and I ended up drawing a diagram to help me traverse the board and check for consecutive sequences of 4.





    - To check if there's a tie-a board being completely full: I created another function that checks if every single column in the board was full by using a for loop and running through each column and checking whether that column was full (using a previous function made to validate input)

- Minimax - minimax(board,depth,maximizingPlayer, alpha,beta)
    - I followed the advice given to me when I mentioned I had 0 experience with AI and looked around for minimax and the general algorithm to understand it better. I took a look at the pseudocode on Wikipedia (looked around at the tic tac toe implementation as well) and tried to interpret and understand it to apply to connect four. In addition I applied alpha and beta pruning to speed up the decision making.
        - Nodes and recursive calculations: I learnt that basically minimax would allow you to basically look ahead into the future (depending on the depth) and decide the best move, assuming that the other player takes the best moves for themselves as well simulating a back and forth, the maximising player would take the board



that maximises their gain/potential for winning, and the minimizing player/opposing player would choose the move that would minimize the opposing player's score. So I created a list of valid moves by creating a function that generates a list of columns the user may put their disc into.

- Minimax would then, depending on whether they were maximising the player or minimizing the player, would look at all possible boards and choose either the worst or the best board by evaluating it and giving it a score. I created a variable to keep track of which was the best column or worst column to use as a move
    - This used recursion and I had to include a base case (terminal node, else just return score) and recursive cases (minimizing and maximising player)

- Evaluating and giving a board a score: to do this I looked through all possible subsequences of length 4 diagonally, vertically, and horizontally and gave those a score as well and summed them up to give the board a total score.
    - Score wise if the board had a sequence of 4 of the player's disk it would give it a very very high score, and good scores for sequences of 3, and bad ones for the opposing player may win after
        - With this algorithm it allows me to take into account the minimax trying to create sequences as well where it forces a win
    - I also had to make the AI favour taking center columns because it opens up the most opportunity to take offensive moves. I noticed this in particular after playing games since It allowed me to do much more diagonal offensive moves, as well as when given the chance to take the first move, it should choose the center column.
- Alpha beta pruning: This would allow me to make the AI faster when doing higher levels of depth. It essentially just compares and optimizes by lowering the number of calculations needed by not performing calculations of certain branches and breaking out of the loop.

- Playing the game:
    - I just randomized the number to make someone the first playing by using random module to choose between 1 and 2

        - I added a random seed using system time to ensure randomness, though I'm not 100% it makes much of a difference
    - Using the minimax I could now use difficulty as a measure of depth as well.
        - Dead easy: completely random moves, the bot is not analysing the board
        - Medium: depth of 1, it just tries to maximise its move but doesn't look really far into the future
        - Relatively hard: depth of 4, looks into the future
    - The user can also play against their friend, in which case the game would just go back and forth with user inputs instead of having an ai play against the user
        - In order to do this I just used some basic input validation and a conditional to set something to true and pass it through the play game function

BUGS AND ERROR/STUFF I LEARNT

- User input validation: I had to just double check the user's input and try to make it stop the user from entering things not within range/full columns by while looping and breaking
- Checking through the diagonals: wasn't a huge issue but it was very mind-bending trying to think of the best way to loop through and traverse through the board correctly.
- Copy vs deep copying lists of lists
    - This was actually one of the biggest issues I ran into while trying to implement my solution for minimax. While I was trying to run the minimax function and test it out, it kept changing the values in my original board, even though I was copying it. That's when I did a little research and it turns out I have to use deep copy instead from the copy module since memory locations of the lists inside the lists were not getting changed even though I copied the first layer of elements (which were lists). I had to use deep copy which would recursively look into elements and copy them so they would have their own memory space for minimax.

3. Provide runtime/space complexity analysis of your solutions. You don't have to analyze every single part but should provide analysis for the main components.

* Here are most of the important functions I used. Also I removed the parameters for functions since it was getting annoying to type (I am aware it technically means it refers to different functions, but note I did not name any functions the same name).

Judy Wu
101184803

Most of my functions were O(n) or O(n^2), except for minimax:

- O(n): these functions
  - addDisc()
    - It just while loops through columns to add a disk
  - colIsFull()
    - It just while loops through columns to count if the column is full
- O(n^2): these functions go through 2 for loops to check every element in the board
  - boardIsFull()
    - Is O(n) however it calls colIsFull() as well which is also O(n) inside the loop, making it O(n^2)
  - initboard()
    - Goes through 2 for loops to create a board and appends an element
  - validMoves()
    - Is O(n) but uses colIsFull() which is is also O(n) inside the loop, making it O(n^2)
  - printboard()
    - Goes through every single element in the board and prints out the element
  - Checks
    - horizontalCheck
    - Vertical check
    - ascDiagonalCheck
- checkWin()
    - O(n^2): Goes through horizontal check, vertical, and diagonal. Horizontal and vertical is o(n^2), but diagonal is o(n^2) since it is actually running asc and desc vertical check which are both o(n^2)
- minmax()
  - Worst case complexity is the complexity of minimax at a depth of 4.
    - Minimax is probably the function that takes up the most complexity since it's recursive as well. Since it has 7 max possible moves at a time, and a depth of 4, considering depth as 'n', b would be the valid moves at the time, this is o(b^n). Even though I implemented alpha beta pruning, in the worst case nothing gets pruned. I'm not sure the exact complexity, but I know for sure it is the most complex function in terms of time and space since it recursively copies the board to test out potential moves.
    - It calls score() which has a complexity of o(n^2)

Some of the complexity changes depending on what the user selects. As an example depth is the biggest issue where I had to code minimax, since the higher your depth the better the ai can make decisions.

4. Provide discussion of any trade offs you considered in implementing your solution. Provide justification for the choices you made.

Originally I was going to create a discord bot using my logic. However I ended up not doing that since I wanted to focus on the computational aspects of my project first and foremost and my ai player since its acc computation rather than me sifting through documentation that I'm completely unfamiliar with like looping through commands etc

Additionally I feel as though I could've made my program less complex in the playGame() function, though I wasn't sure about what areas to improve on specifically. For other functions like checking win or minimax, I felt like you had to do a lot of calculations and traversing through the board and I wasn't exactly sure what I could do to make it less intensive other than alpha and beta pruning.

5. Discuss any extensions, improvements, or future work that you have considered but not implemented. If you have thought about any in detail, you can also provide some algorithm details.

- Better User Interface:
    - I ended up not being able to display emojis in console and did not port to discord. Definitely coding using discord and their documentation and figuring out how to loop commands is something I plan to do so I can play with my friends on my server. I took an attempt at it and greatly underestimated how different it was compared to coding and outputting things in a console.
    - Another thing I could make as an extension is using pygame to output the game instead visually
- Better AI
    - I definitely think I could have tweaked my AI a bit more and have it score pieces closer to the bottom of the board better since in the early game it would open up more opportunity for creating 4 consecutive sequences.
- Better validation
    - Even though I implemented validation for whether the user would enter something out of bounds from accepted numbers, or whether a column is full, I did not figure out how to prevent the user from entering any other data type which would break my code :(