PROJECT TEAM NAME

Team X

PROJECT NAME

CONNECT4

|COURSE NAME|

SWE-150 PROJECT WORK I

- |Course teacher|

Dr. Ahsan Habib

MEET OUR TEAM

MAHIR AL SHAHRIAR

REG-2019821077

SAKIBUL ISLAM

REG-2019821062

SABBIR HOSSIN

REG-2019821037

NAIMUR RUMEL

REG-2019821079

ZAWADUL KARIM

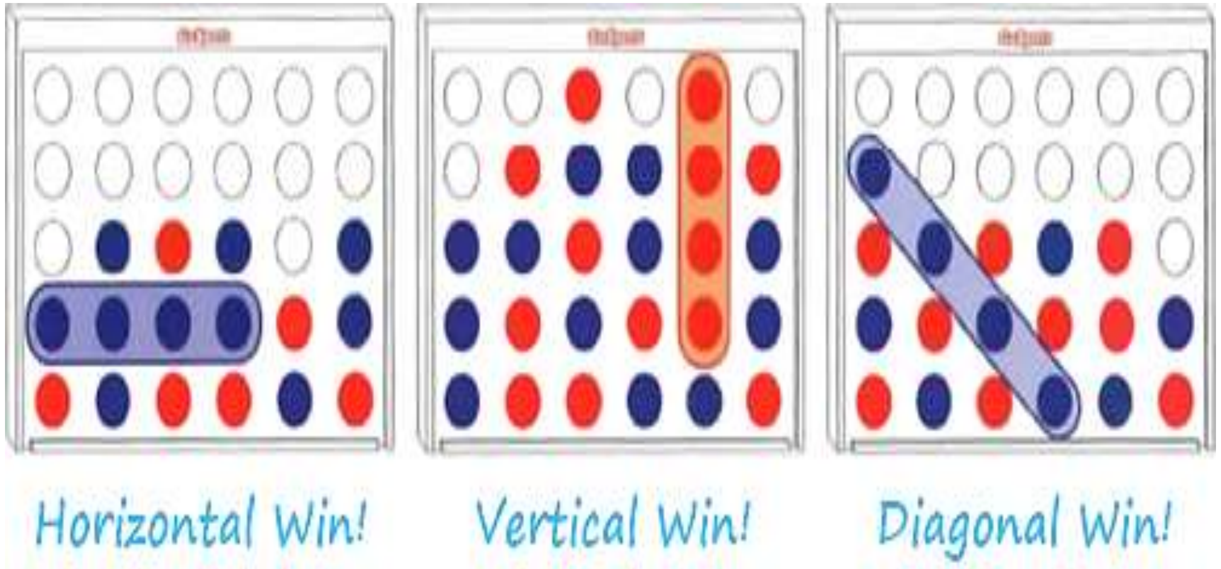
REG-2019821064

INTRODUCTION TO CONNECT4

Connect-Four is a tic-tac-toe-like two-player game in which players alternately place pieces on a vertical board 7 columns across and 6 rows high. Each player uses pieces of a particular color (commonly black and red, or sometimes yellow and red), and the object is to be the first to obtain four pieces in a horizontal, vertical, or diagonal line. Because the board is vertical, pieces inserted in a given column always drop to the lowest unoccupied row of that column. As soon as a column contains 6 pieces, it is full, and no other piece can be placed in the column.

HOW PLAY TO CONNECT4

- It's You (Red Coins vs the Computer (Blue Coins2)
- Players take turn to drop coins. Click top to drop
- To win, you must form a line of 4 red coins. The line can be vertical, horizontal or diagonal



❑ Modeling the game logic-the MAIN FUNCTION

❖ MODELLING GAME DATA:

- ➢ THE GAME BOARD: MATHEMATICALLY :- 6X7 MATRIX: COMPUTATIONALLY 2D ARRAY BOARD
- ➢ PLAYER INFORMATION: MATHEMATICALLY: PAIR(2-TUPLE) CONSISTING OF NAME AND SYMBOL: COMPUTATIONALLY WE USED THE STRUCTURE PLAYER WITH MEMBERS: NAME- STRING, AND SYMBOL: CHARACTER VARIABLE
- ➢

  RESULT: MODELLED INTO NUMERICAL DATA 0,1,2 TO REPRESENT CONTINUATION, WINNING, AND DRAW POSITION, USING VARIABLE RESULT

MAIN FUNCTION

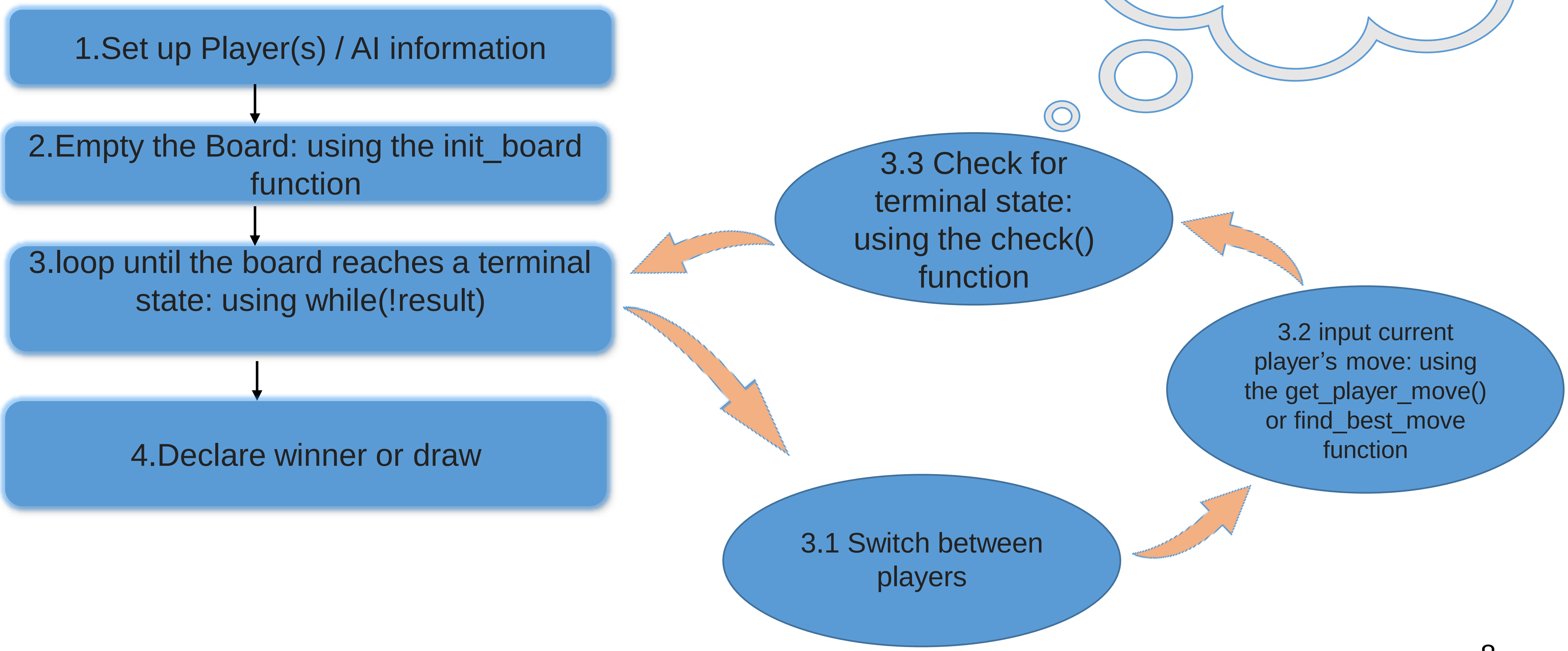
AND SLIDE PRESENTATION

BY ZAWADUL KARIM

REG NO: 2019831064

❑ MODELLING THE GAME PROCESS:

THE PROGRAM FLOWS AS FOLLOWS:



1.Set up Player(s) / AI information

2.Empty the Board: using the init_board function

3.loop until the board reaches a terminal state: using while(!result)

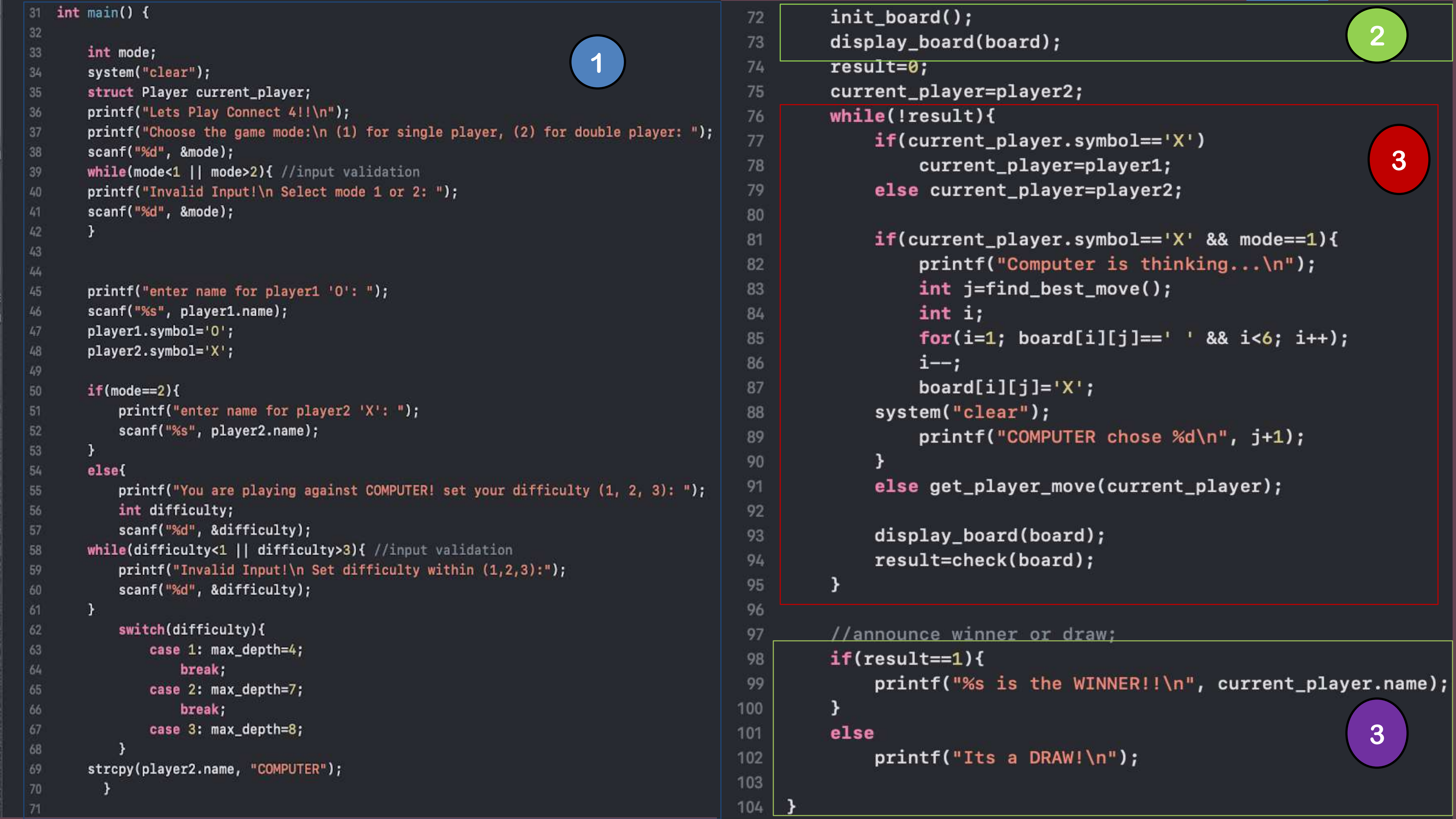
4.Declare winner or draw

3.1 Switch between players

3.2 input current player’s move: using the get_player_move() or find_best_move function

3.3 Check for terminal state: using the check() function

Throughout these execution loop use display_board function and system(“clear”) call to display and manage the outlook of CLI



```c
int main() {

    int mode;
    system("clear");
    struct Player current_player;
    printf("Lets Play Connect 4!!\n");
    printf("Choose the game mode:\n (1) for single player, (2) for double player: ");
    scanf("%d", &mode);
    while(mode<1 || mode>2){ //input validation
    printf("Invalid Input!\n Select mode 1 or 2: ");
    scanf("%d", &mode);
    }


    printf("enter name for player1 'O': ");
    scanf("%s", player1.name);
    player1.symbol='O';
    player2.symbol='X';

    if(mode==2){
        printf("enter name for player2 'X': ");
        scanf("%s", player2.name);
    }
    else{
        printf("You are playing against COMPUTER! set your difficulty (1, 2, 3): ");
        int difficulty;
        scanf("%d", &difficulty);
    while(difficulty<1 || difficulty>3){ //input validation
        printf("Invalid Input!\n Set difficulty within (1,2,3):");
        scanf("%d", &difficulty);
    }
        switch(difficulty){
            case 1: max_depth=4;
                break;
            case 2: max_depth=7;
                break;
            case 3: max_depth=8;
        }
    strcpy(player2.name, "COMPUTER");
      }

    init_board();
    display_board(board);
    result=0;
    current_player=player2;
    while(!result){
        if(current_player.symbol=='X')
            current_player=player1;
        else current_player=player2;

        if(current_player.symbol=='X' && mode==1){
            printf("Computer is thinking...\n");
            int j=find_best_move();
            int i;
            for(i=1; board[i][j]==' ' && i<6; i++);
            i--;
            board[i][j]='X';
        system("clear");
            printf("COMPUTER chose %d\n", j+1);
        }
        else get_player_move(current_player);

        display_board(board);
        result=check(board);
    }

    //announce winner or draw;
    if(result==1){
        printf("%s is the WINNER!!\n", current_player.name);
    }
    else
        printf("Its a DRAW!\n");

}
```

CHECK FUNCTION

BY SAKIBUL ISLAM

REG NO: 2019831062

Check Function

The check() function [Int check(char board[6][7])] takes the board(2d char array) as input and returns 1 for win condition, 2 for draw condition, 0 for continuation. For win condition, this function checks if four continuous piece of same type is found on:

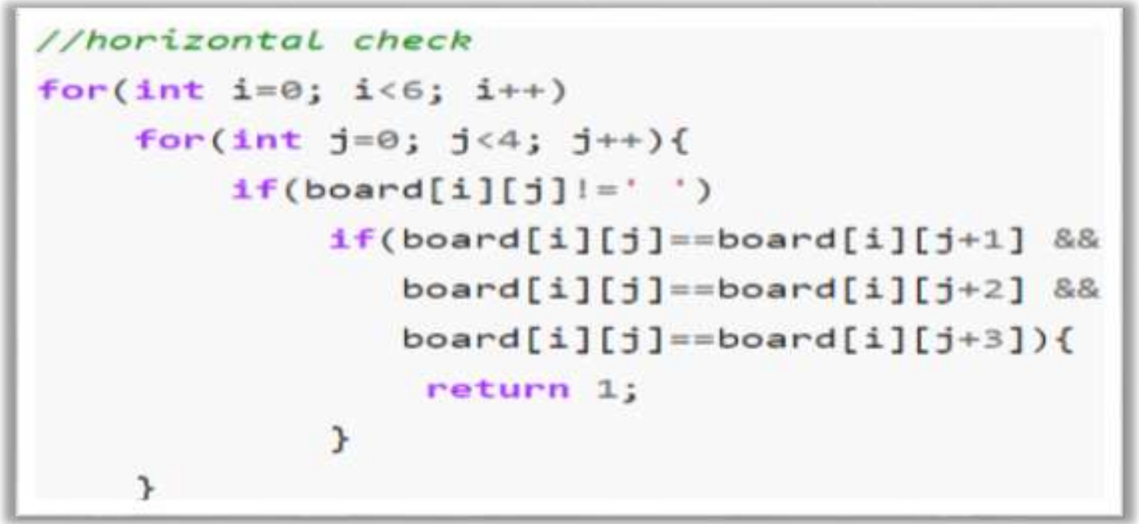
- Horizontal line or
- Vertical line or
- Diagonal line.

So, the check() function implemented various checks to check for each of these conditions as well as a full board condition. If the win conditions are not true, the function checks for full board condition. Full board condition checks if all the cells in the board is fulfilled. If this condition is true, the check function returns 2 indicating a draw. If the win conditions are not true and the board is not fulfilled, the function returns 0 indicating the continuation of the game.

Horizontal Check

In this section, the function search for four continuous same piece in a row. In each of the six rows, if one cell is non-empty (meaning it contains one kind of a piece), the function checks whether next three cell of that row also contains that same piece. Such as, if the 4th cell of one row contains (X), the function search if the 5th, 6th and 7th cells also contains (X).

To optimize the code, the function doesn't check further if a piece is found at 5th cell of a row. Because, even if the 6th and 7th cells of that row contains the same piece, it won't fulfill the criteria of matching four continuous same piece for a horizontal win. This is due to the board containing only 7 columns. So, checking the horizontal win condition for the first four column in each row is sufficient.

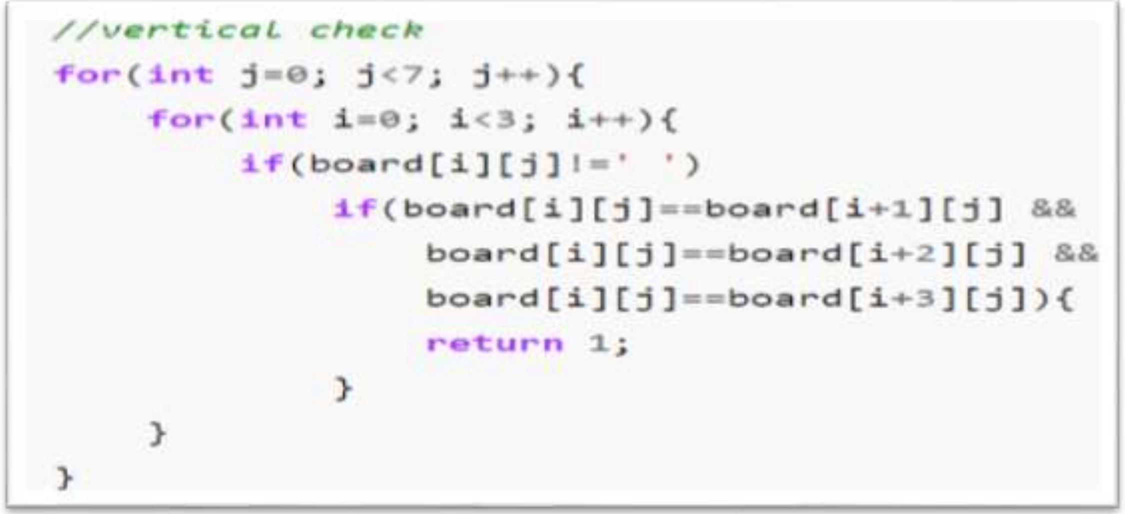


```
//horizontal check
for(int i=0; i<6; i++)
    for(int j=0; j<4; j++){
        if(board[i][j]!=' ')
            if(board[i][j]==board[i][j+1] &&
               board[i][j]==board[i][j+2] &&
               board[i][j]==board[i][j+3]){
                return 1;
            }
    }
```

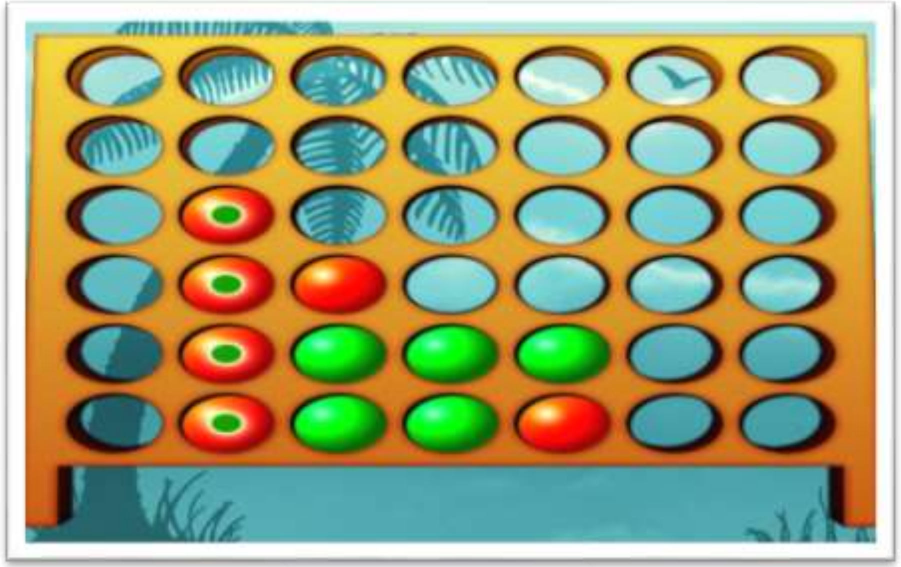
Vertical Check

In this section, the function search for four continuous same piece in a column. In each of the seven columns, if one cell is non-empty, the function checks whether next three cells of that column contains the same piece. Such as, if the 3rd cell of one column contains (X), the function search if the 4th, 5th and 6th cells also contains (x).

To optimize the code, the function doesn't check further if a piece is found at 4th cell of a column. Because, even if the 5th and 6th cells of that column contains the same piece, it won't fulfill the criteria of matching four continuous same piece for a vertical win. This is due to the board containing only 6 rows. So, checking the vertical win condition for the first three row in each column is sufficient.



```
//vertical check
for(int j=0; j<7; j++){
    for(int i=0; i<3; i++){
        if(board[i][j]!=' ')
            if(board[i][j]==board[i+1][j] &&
                board[i][j]==board[i+2][j] &&
                board[i][j]==board[i+3][j]){
                return 1;
            }
    }
}
```



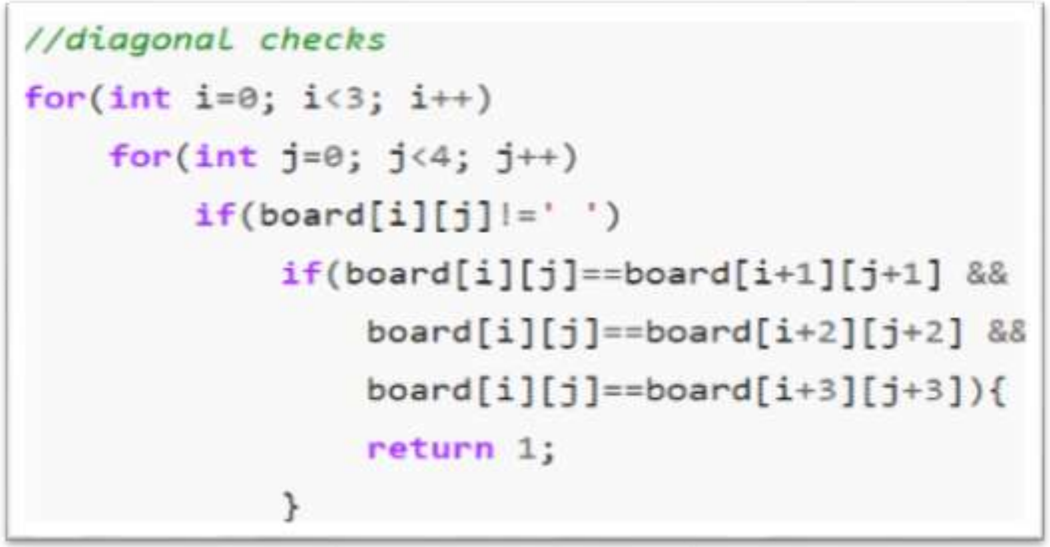
Diagonal Check
(Upper-Left to Lower-Right)

In this section, the function search for four continuous same piece diagonally from left to right. For this, the condition checks if the immediate bottom right cell of a piece contains the same piece. This type of diagonal four cells containing same piece should be present for a diagonal win condition to be true. To form a diagonal of length 4, we need 4 valid rows and 4 valid columns. If a piece (x) is found on 4th column of 3rd row (3, 4)cell, the next three (X) pieces should be on (4, 5), (5, 6) and (6, 7) cells.

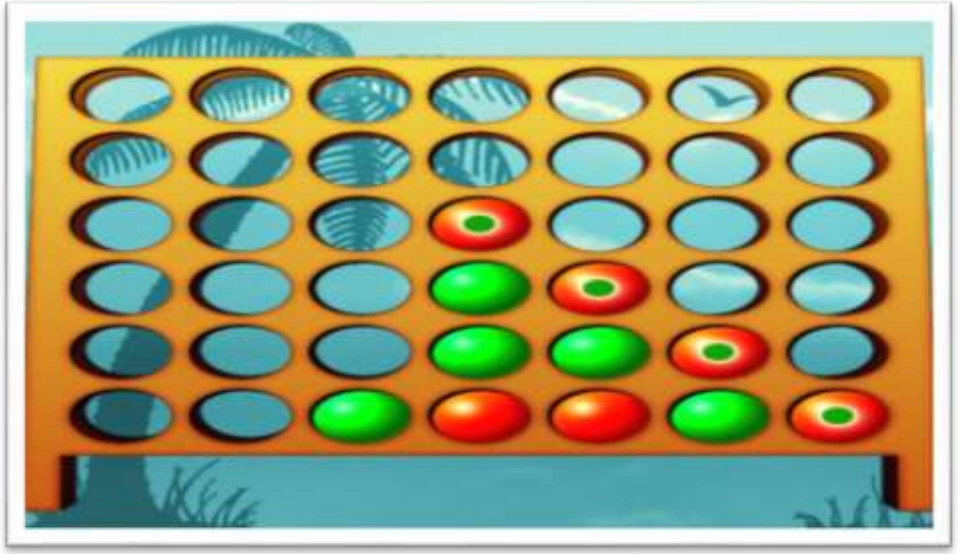
To optimize the code, the function only checks the first 3 rows and first 4 columns. If a piece is found at 4th row, maximum 3 pieces of same type can be found as only 5th and 6th row is available. But to form a diagonal, 4 rows are required. So, checking the diagonal condition for first 3 rows is sufficient.

Again, if a piece is found at 5th column, maximum 3 pieces of same type can be found as only 6th and 7th column is available. But to form a diagonal, 4 columns are required. So, checking the diagonal condition for first 4 columns is sufficient.

So, for a diagonal win from upper left to lower right, the condition is checked for only first 3 rows and first 4 columns.



```
//diagonal checks
for(int i=0; i<3; i++)
    for(int j=0; j<4; j++)
        if(board[i][j]!=' ')
            if(board[i][j]==board[i+1][j+1] &&
                board[i][j]==board[i+2][j+2] &&
                board[i][j]==board[i+3][j+3]){
                return 1;
            }
```



Diagonal Check

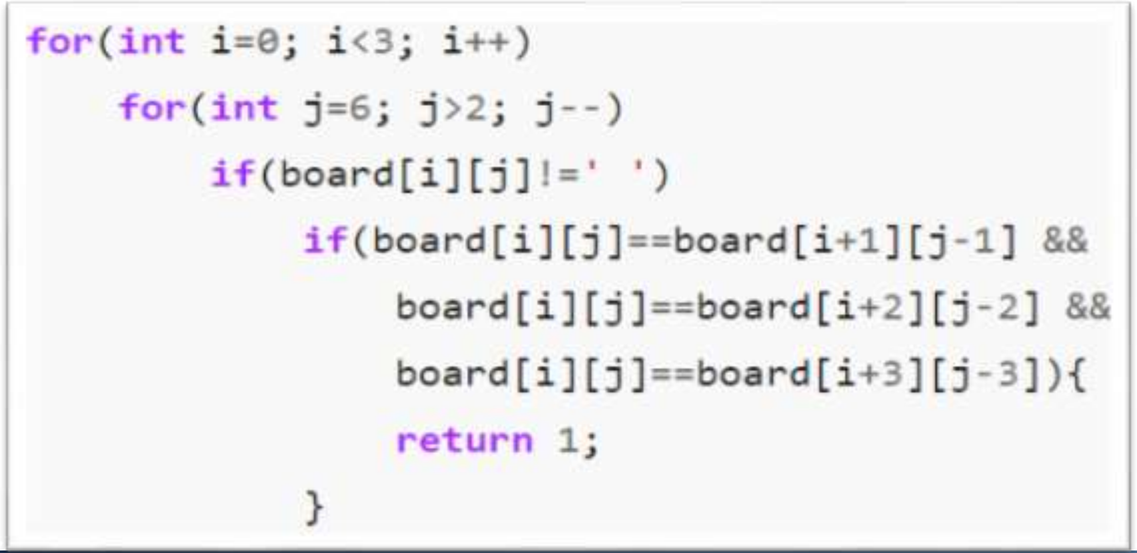
(Upper-Right to Lower-Left)

In this section, the function search for four continuous same piece diagonally from right to left. For this, the condition checks if the immediate bottom left cell of a piece contains the same piece. This type of diagonal four cells containing same piece should be present for a diagonal win condition to be true. To form a diagonal of length 4, we need 4 valid rows and 4 valid columns. If a piece (x) is found on 4th column of 3rd row (3, 4)cell, the next three (X) pieces should be on (4, 3), (5, 2) and (6, 1) cells.

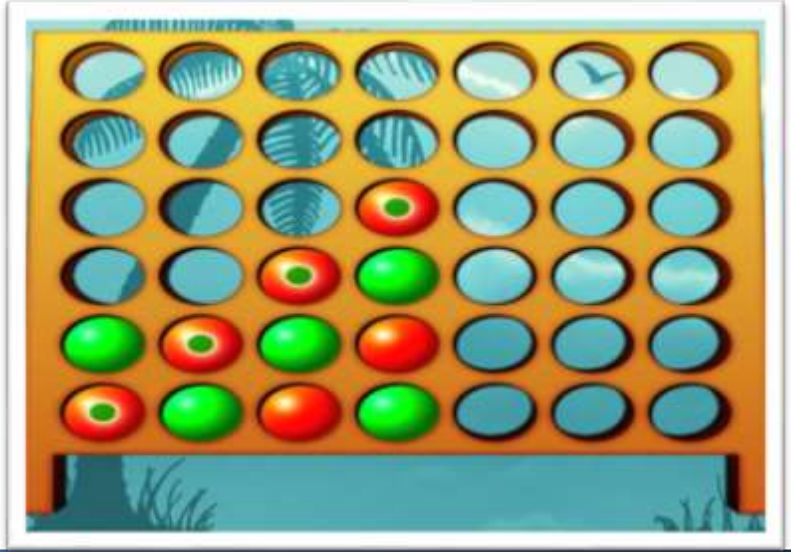
To optimize the code, the function only checks the first 3 rows and last 4 columns. If a piece is found at 4th row, maximum 3 pieces of same type can be found as only 5th and 6th row is available. But to form a diagonal, 4 rows are required. So, checking the diagonal condition for first 3 rows is sufficient.

Again, if a piece is found at 3rd column, maximum 3 pieces of same type can be found as only 1st and 2nd column is available. But to form a diagonal, 4 columns are required. So, checking the diagonal condition for last 4 columns is sufficient.

So, for a diagonal win from upper right to lower left, the condition is checked only for first 3 rows and last 4 columns.

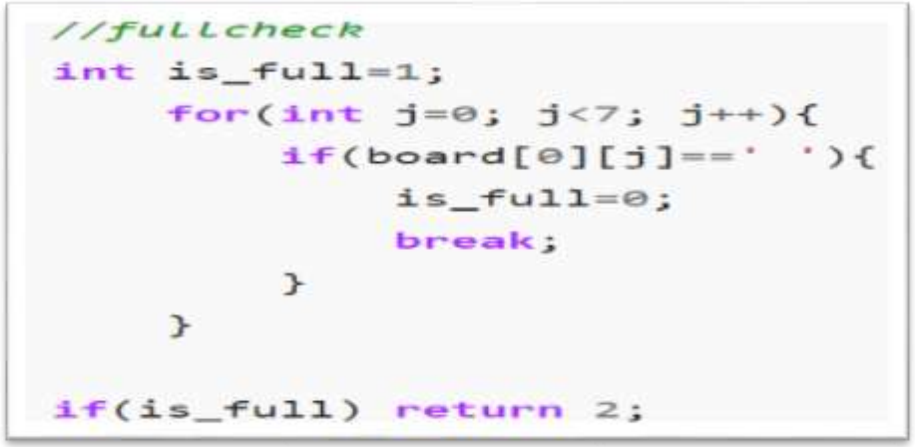


```
for(int i=0; i<3; i++)
    for(int j=6; j>2; j--)
        if(board[i][j]!=' ')
            if(board[i][j]==board[i+1][j-1] &&
                board[i][j]==board[i+2][j-2] &&
                board[i][j]==board[i+3][j-3]){
                return 1;
            }
```



Full Board Check

To check if the board is fulfilled, the functions doesn’t need to check every cell whether it is fulfilled. As the board is vertical, pieces inserted in each column always drops to the lowest unoccupied row of that column. As soon as a column contains 6 pieces, it is full, and no other piece can be placed in the column. The board has 7 columns. If all the columns are full, the board is full and for a column to be fulfilled, its top row needs to be fulfilled. So, the function only checks if the topmost cell of every column is fulfilled.



```
//fullcheck
int is_full=1;
    for(int j=0; j<7; j++){
        if(board[0][j]==' '){
            is_full=0;
            break;
        }
    }

if(is_full) return 2;
```

INIT BOARD &
DISPLAY BOARD

BY MD. SABBIR HOSSIN

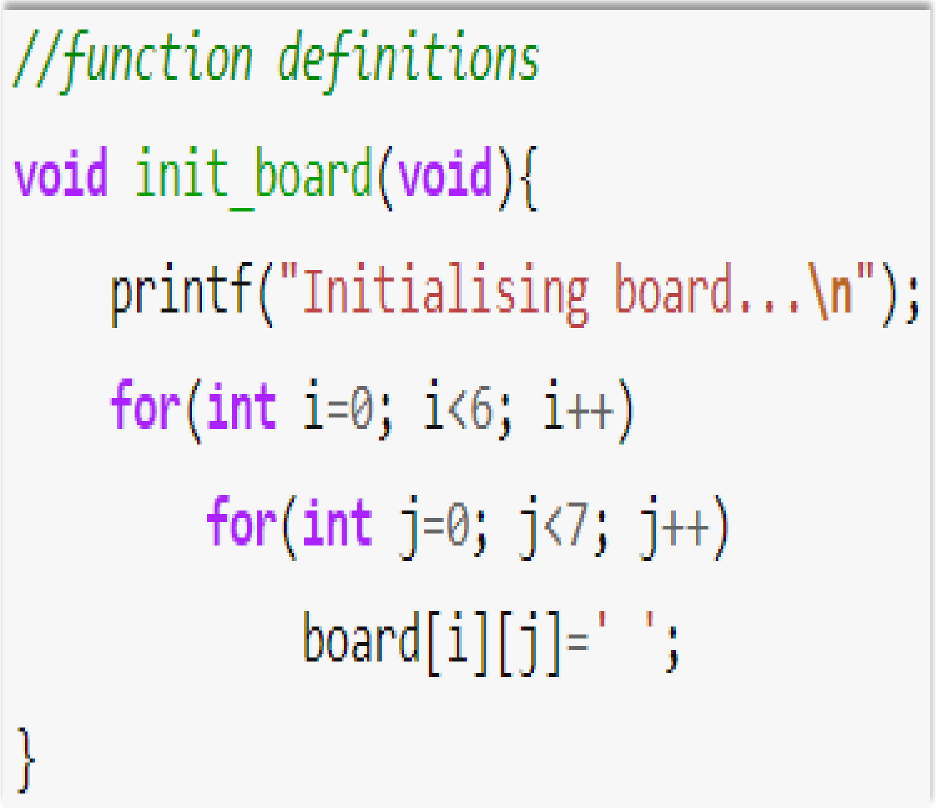
REG NO: 2019831037

INIT BOARD

1. The outer loop traverse's rows 0-5 using index i .

2. For each iteration of the outer loop, the inner loop traverse's columns 0-6 using the index j.

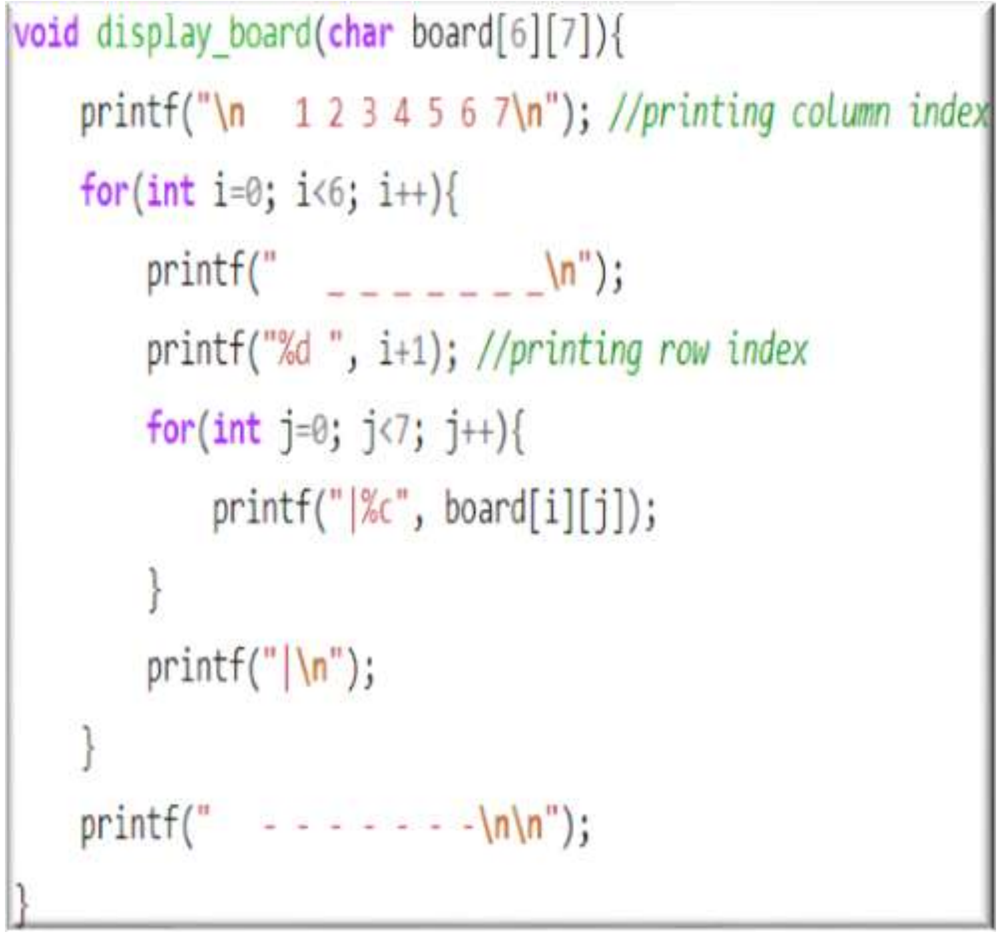
3. For every pair (i, j) the corresponding entry is set to space ' '.



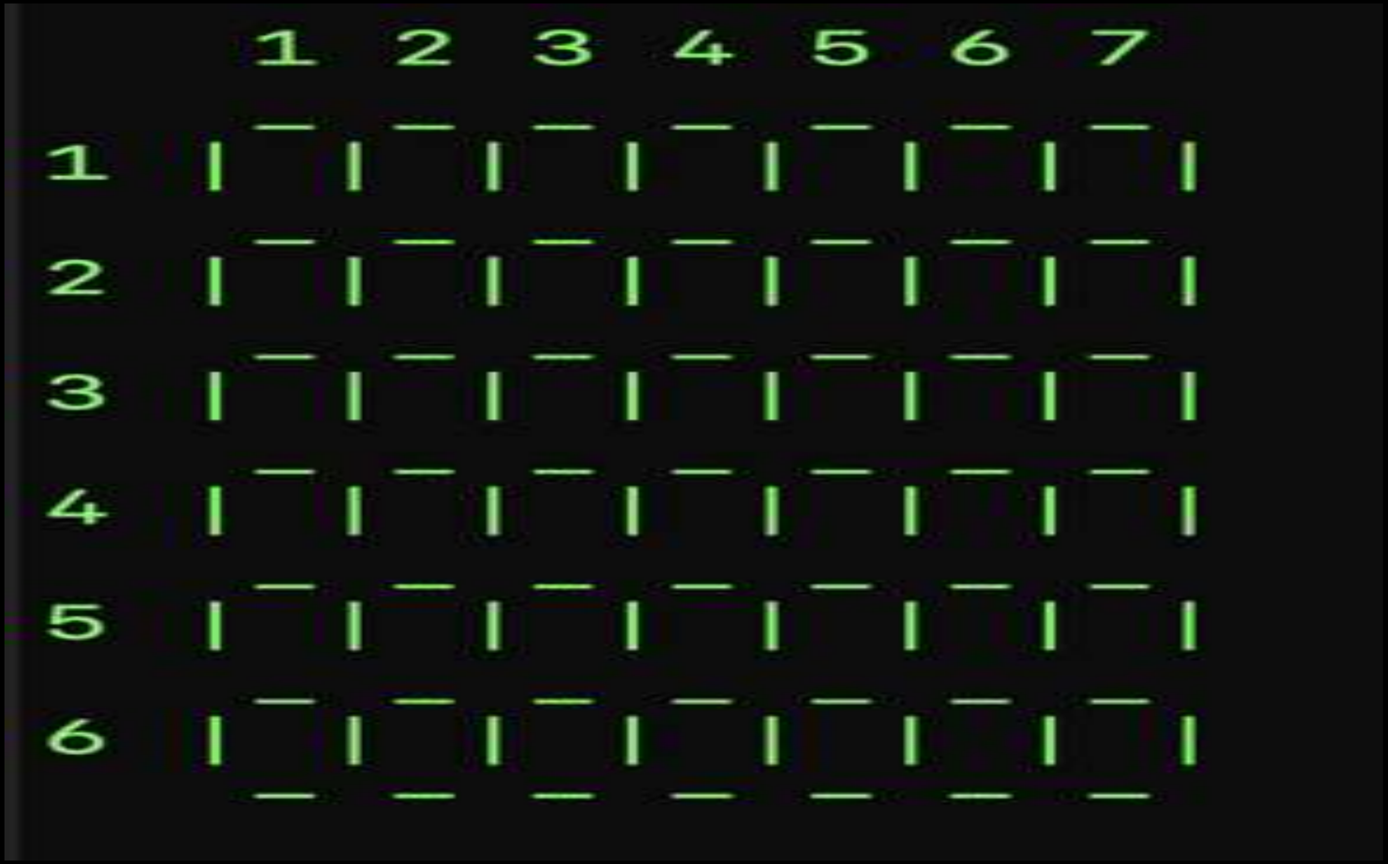
```
//function definitions
void init_board(void){
    printf("Initialising board...\n");
    for(int i=0; i<6; i++)
        for(int j=0; j<7; j++)
            board[i][j]=' ';
}
```

DISPLAY BOARD

1. Printing column index in a single line.
2. Loop through the rows and print the roofs using _ underscores.
3. For each row print the row index and column entries separated by '|' by using an inner loop.
4. Finally the floor is made using hyphens (-).



```
void display_board(char board[6][7]){
    printf("\n   1 2 3 4 5 6 7\n"); //printing column index
    for(int i=0; i<6; i++){
        printf("   _ _ _ _ _ _ _\n");
        printf("%d ", i+1); //printing row index
        for(int j=0; j<7; j++){
            printf("|%c", board[i][j]);
        }
        printf("|\n");
    }
    printf("   - - - - - - -\n\n");
}
```



```
   1 2 3 4 5 6 7
   _ _ _ _ _ _ _
1 | | | | | | | |
   _ _ _ _ _ _ _
2 | | | | | | | |
   _ _ _ _ _ _ _
3 | | | | | | | |
   _ _ _ _ _ _ _
4 | | | | | | | |
   _ _ _ _ _ _ _
5 | | | | | | | |
   _ _ _ _ _ _ _
6 | | | | | | | |
   _ _ _ _ _ _ _
```

GET PLAYER MOVE()

BY NAIMUR RAHMAN

REG NO: 2019831079

❑ get_player_move()

First of all, when mode is 1, the program will enter into get_player_move() if current player’s symbol is not ‘X’. This indicate It’s the turn of player’s move. (when game is played between player and computer and its player’s turn).Then the program will execute get_player_move().

For another reason, program will execute get_player_move(). When mode is 2, the game is played between two external players (one of them is not computer). For both players get_player_move () will be executed consecutively.

Finally, the program execute get_player_move() which takes column number from current player and keep it in valid cell.

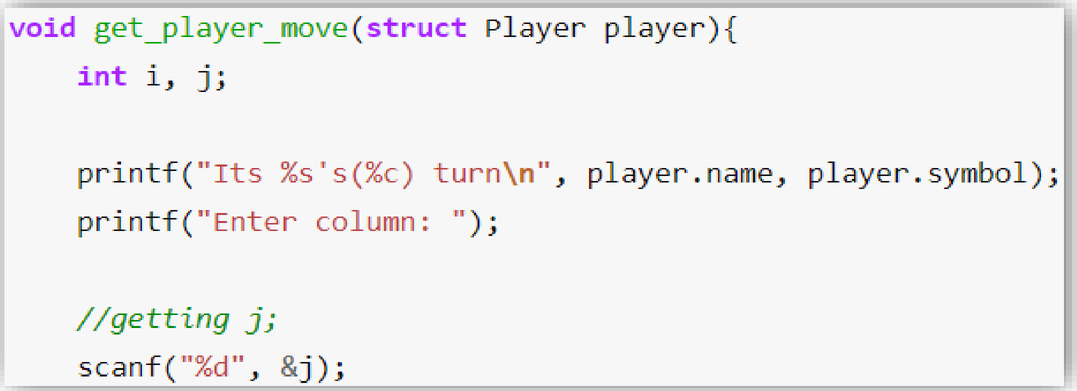
❑ get_player_move()

get_player_move() is a void function, and it receives a structure as a parameter. (player-1/player-2) The parameter is the current player.

We have declared two variable named i and j in get_player_move(). Where i will use to find valid row of player’s chosen column and j will use to take input from current player.

Next the get player move function will print current player Name and current player symbol.

Then it will ask player to provide column number. Then player will provide his willing column number as j.



```c
void get_player_move(struct Player player){
    int i, j;

    printf("Its %s's(%c) turn\n", player.name, player.symbol);
    printf("Enter column: ");

    //getting j;
    scanf("%d", &j);
```

❑ get_player_move()

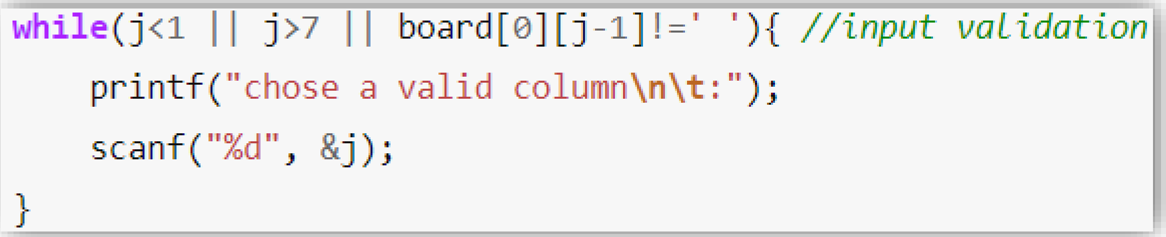
We have to check whether input is valid or not.

For this we have to do two tasks.

We have to check whether the provided value remains between 1 to 7.

The function will also check if the input column has any empty cell or not.

If the provided value is not valid we will repeatedly take input from player and we will check the input validation.



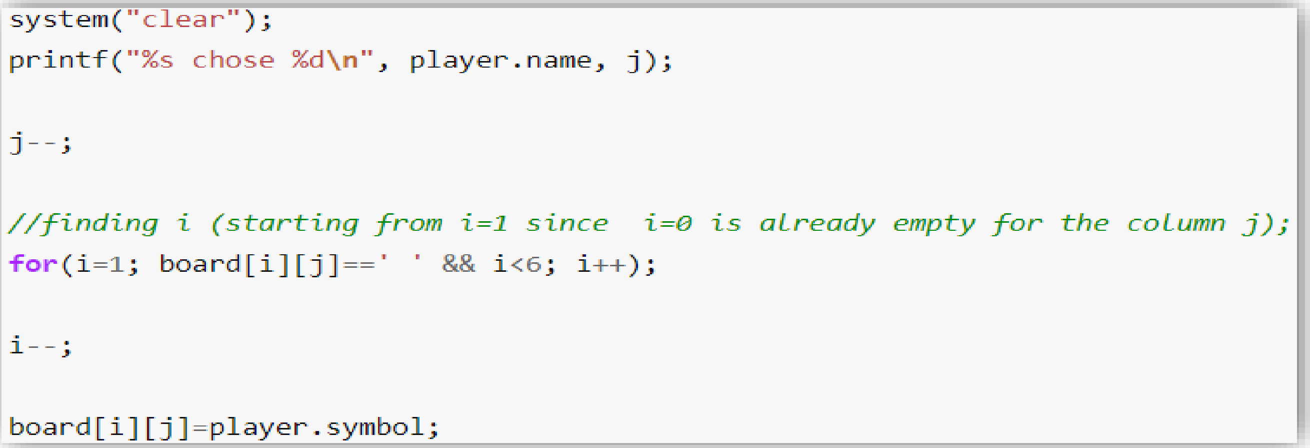
```
while(j<1 || j>7 || board[0][j-1]!=' '){ //input validation
    printf("chose a valid column\n\t:");
    scanf("%d", &j);
}
```

❑ get_player_move()

Next, we will clear terminal window using operating system call system(“clear”). Then we will print current player’s name and his chosen column number.

After that we will decrease 1 from the value of selected column number as array starts counting from 0.

Following this we will find empty row (check row) and we will keep player symbol in that valid row.



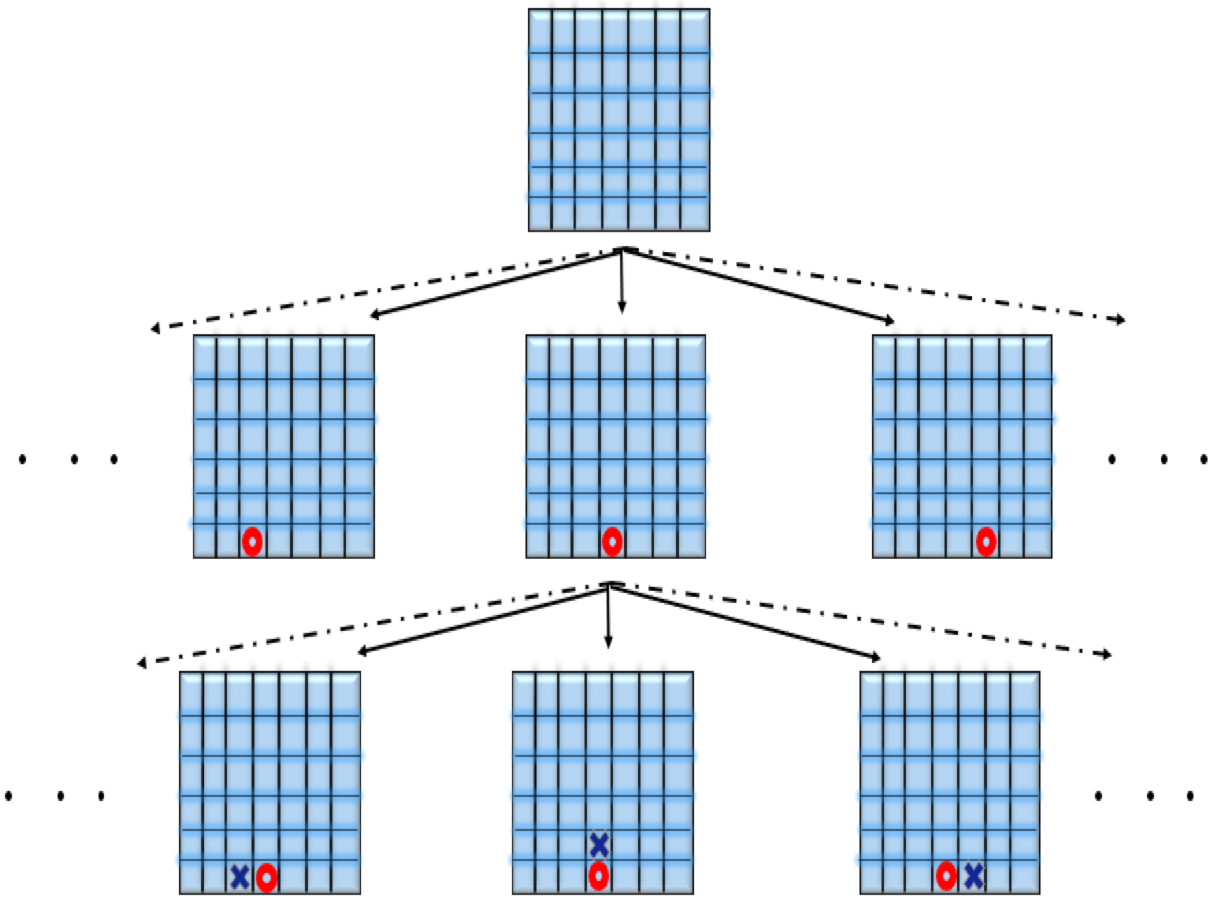
```c
system("clear");
printf("%s chose %d\n", player.name, j);

j--;

//finding i (starting from i=1 since  i=0 is already empty for the column j);
for(i=1; board[i][j]==' ' && i<6; i++);

i--;

board[i][j]=player.symbol;
```

FIND_BEST_MOVE() & MINIMAX() FUNCTIONS

BY MAHIR AL SHAHRIAR
REG NO: 2019831077

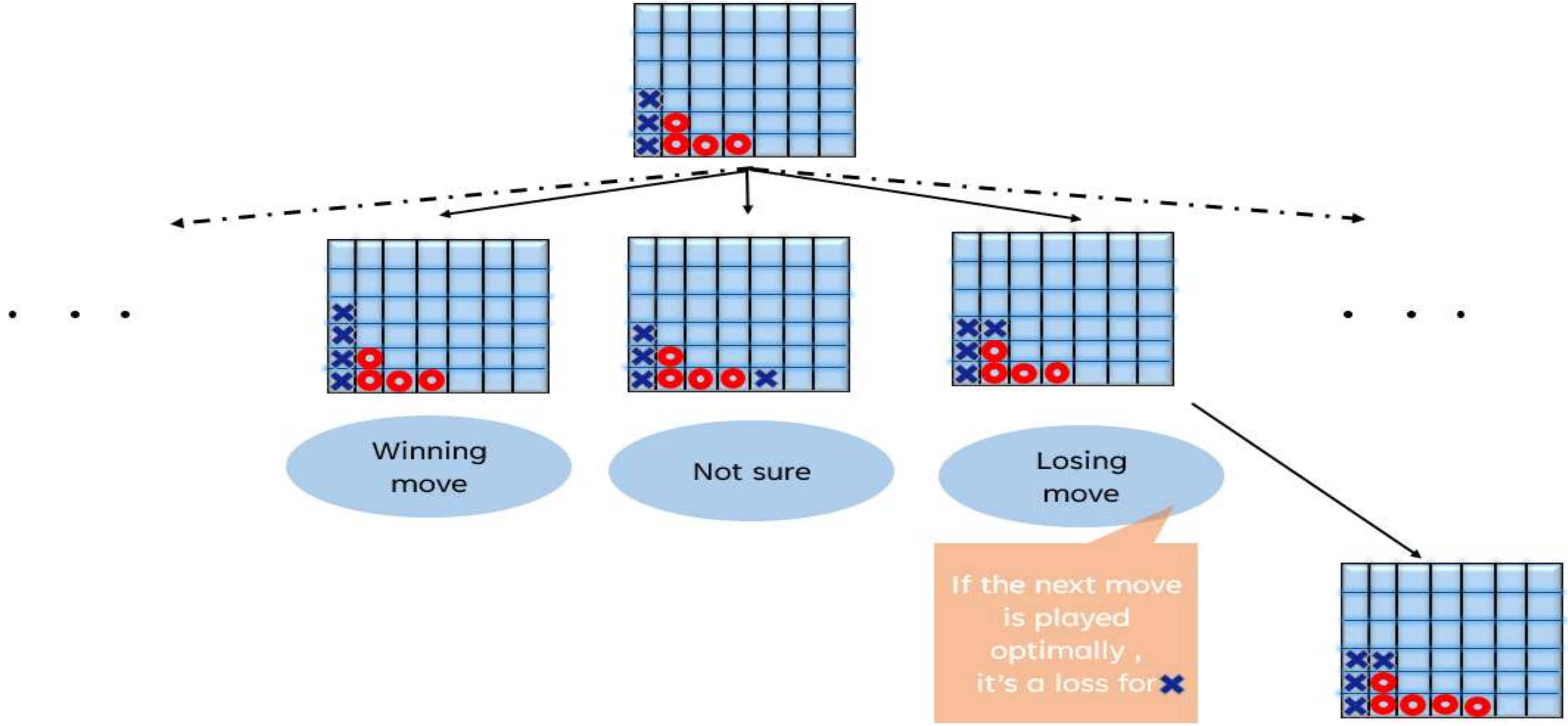
❑ PROGRAMMING THE ARTIFICIAL INTELLIGENCE

- It Helps To Design Our Algorithm For The AI If We Represent The Game Mathematically .
  - ➢ Set of Board Positions P (A board position involves the state of pieces on the board as well as whose turn it is to make the move)
  - ➢ A binary relation on the set P, where (p1,p2) belongs to R iff p2 could be reached from p1 by a single legal move.
- Therefore, the game G=(P,R) denotes a directed graph.



❑ CAPTURING PLAYER BEHAVIOR IN OUR AI:

- ➢ GIVEN A BOARD POSITION THE AI WILL CHOOSE THE OPTIMAL MOVE



Winning move

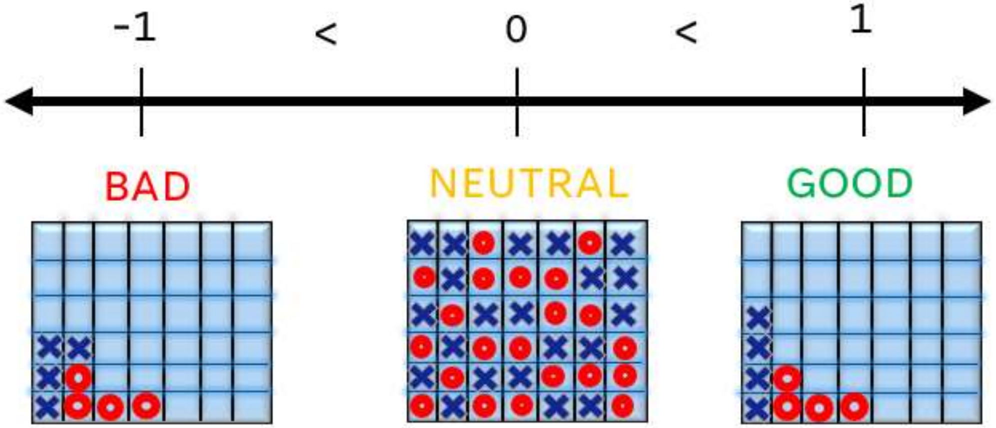
Not sure

Losing move

If the next move is played optimally , it's a loss for ✖

❑CHARACTERIZING AN OPTIMAL MOVE

- We assigned numerical weights to the board positions.
- We used a linear ordering relation on the weights.
  - ➢We used -1 to denote a losing position for the AI
  - ➢0 for a draw position
  - ➢1 for a winning position.
  - ➢> greater than relational operator, to maximize the value (optimal value)



❑ <u>Assigning weights to positions:</u>

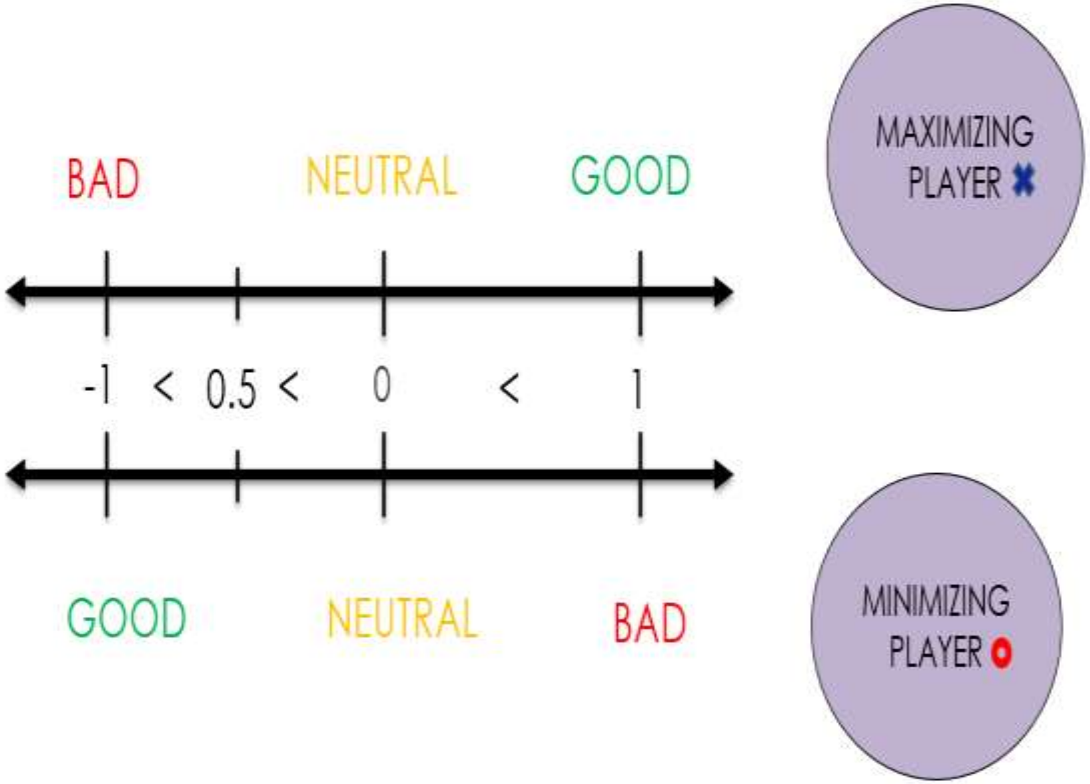
- Terminal state positions will be assigned 1, 0, or -1 depending on if its a loss win or draw.
- As for continuing states, their value would be the value of the final terminal state reached from this position given that both the players played optimally
  - This requires switching perspective for each alternating player, and assigning weights to the next available positions, and choosing the optimal solution according to that players perspective.

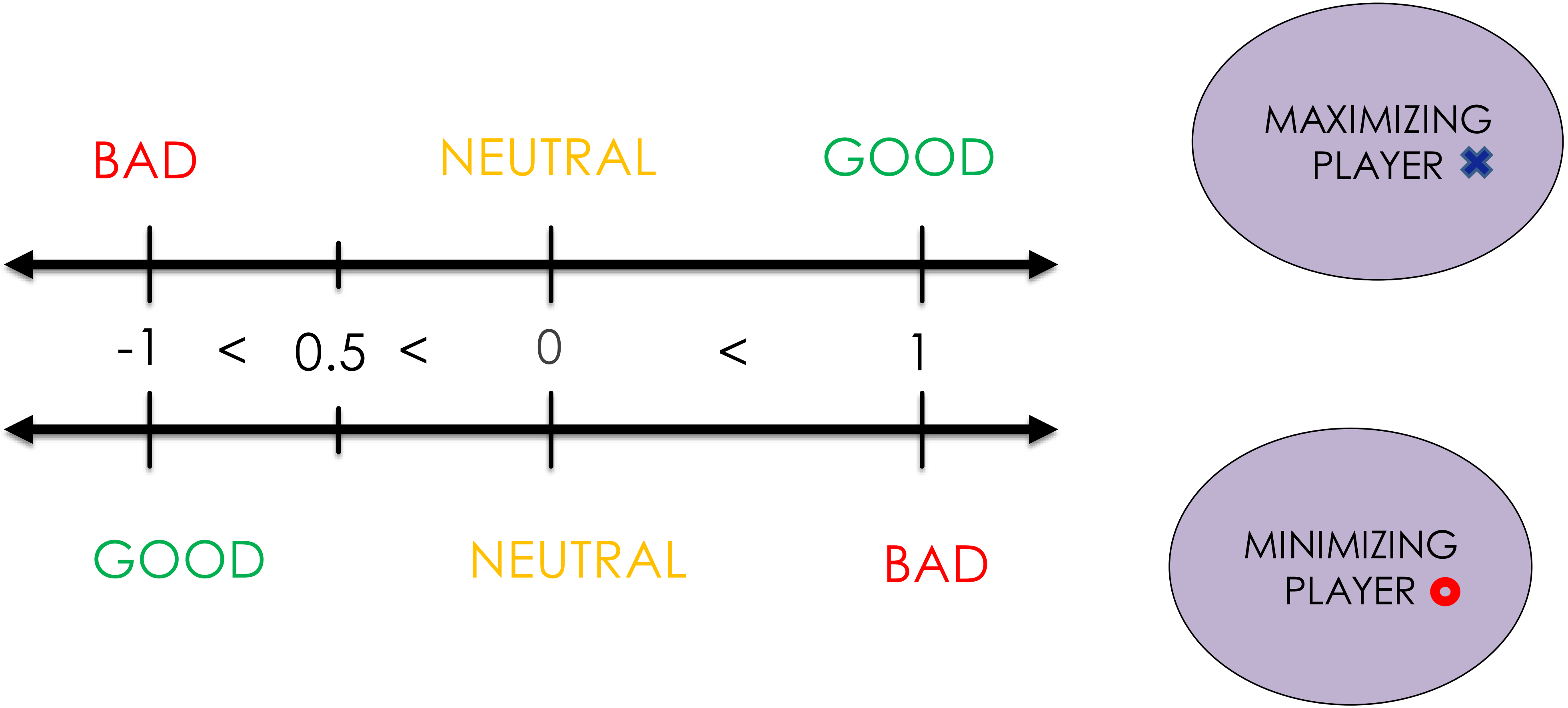
INTROSPECTION…..

- This process of assigning weights calls itself and moves closer to the base cases(terminal states), hence it could be implemented by a recursive function.(we used minimax() in this case).
- Our recursion can be seen as a depth first graph traversal, which ends up in a leaf node (a terminal position).
- Since 4531985219092 different possible positions are there, trying to traverse the graph for each branch would be practically unacceptable.
- Therefore, we include a maximum depth (max_depth) upto which the traversal would continue. If the end position at the max_depth is still in a continuation state, the weight of -0.5 will be assigned.

Switching perspectives

- An optimal move for a player may not be the same for the opponent.
- Hence, we could have taken 2 possible solutions:
  - Either the weights assigned to position could be different for each player (invert their signs for perspective of the player '0': Wo(p1)=-Wx(p1)), but the linear ordering relation kept same.
  - Or keep the assigned weights constant for each position but use different ordering priorities for different perspective.(**This is what we implemented**: < less then relational operator was used for the player 0's perspective)
- Therefore 'X' is called the maximizing player And '0' is called the minimizing player.





BAD NEUTRAL GOOD

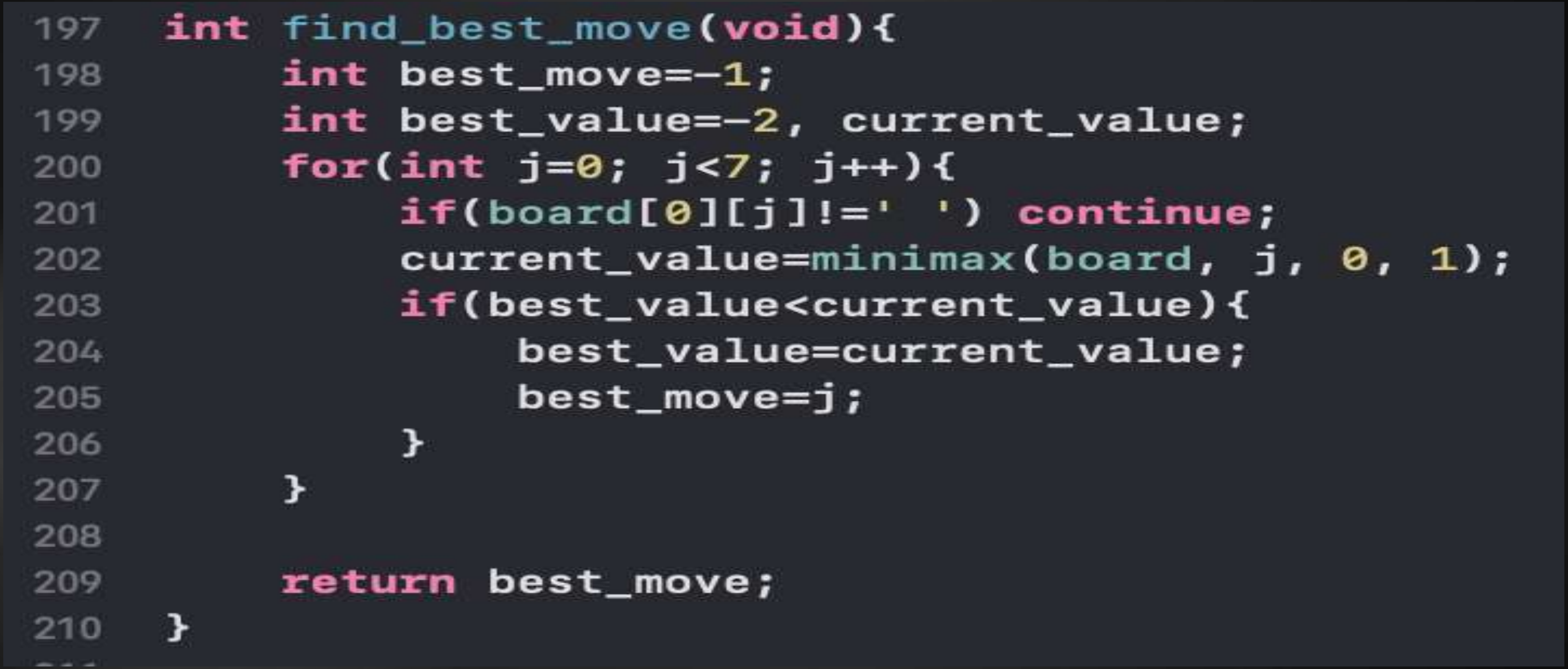
-1 < 0.5 < 0 < 1

GOOD NEUTRAL BAD

MAXIMIZING PLAYER ✖

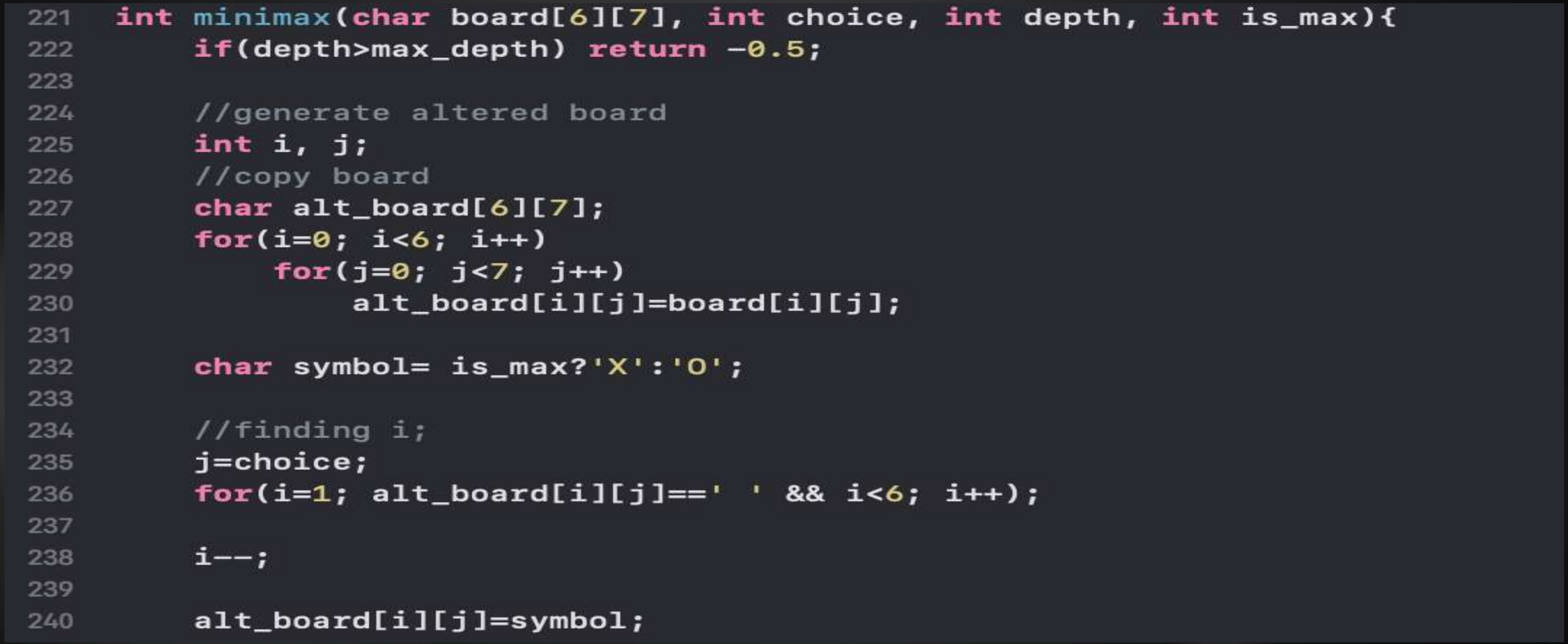
MINIMIZING PLAYER ⊙

find_best_move() function



```c
int find_best_move(void){
    int best_move=-1;
    int best_value=-2, current_value;
    for(int j=0; j<7; j++){
        if(board[0][j]!=' ') continue;
        current_value=minimax(board, j, 0, 1);
        if(best_value<current_value){
            best_value=current_value;
            best_move=j;
        }
    }

    return best_move;
}
```

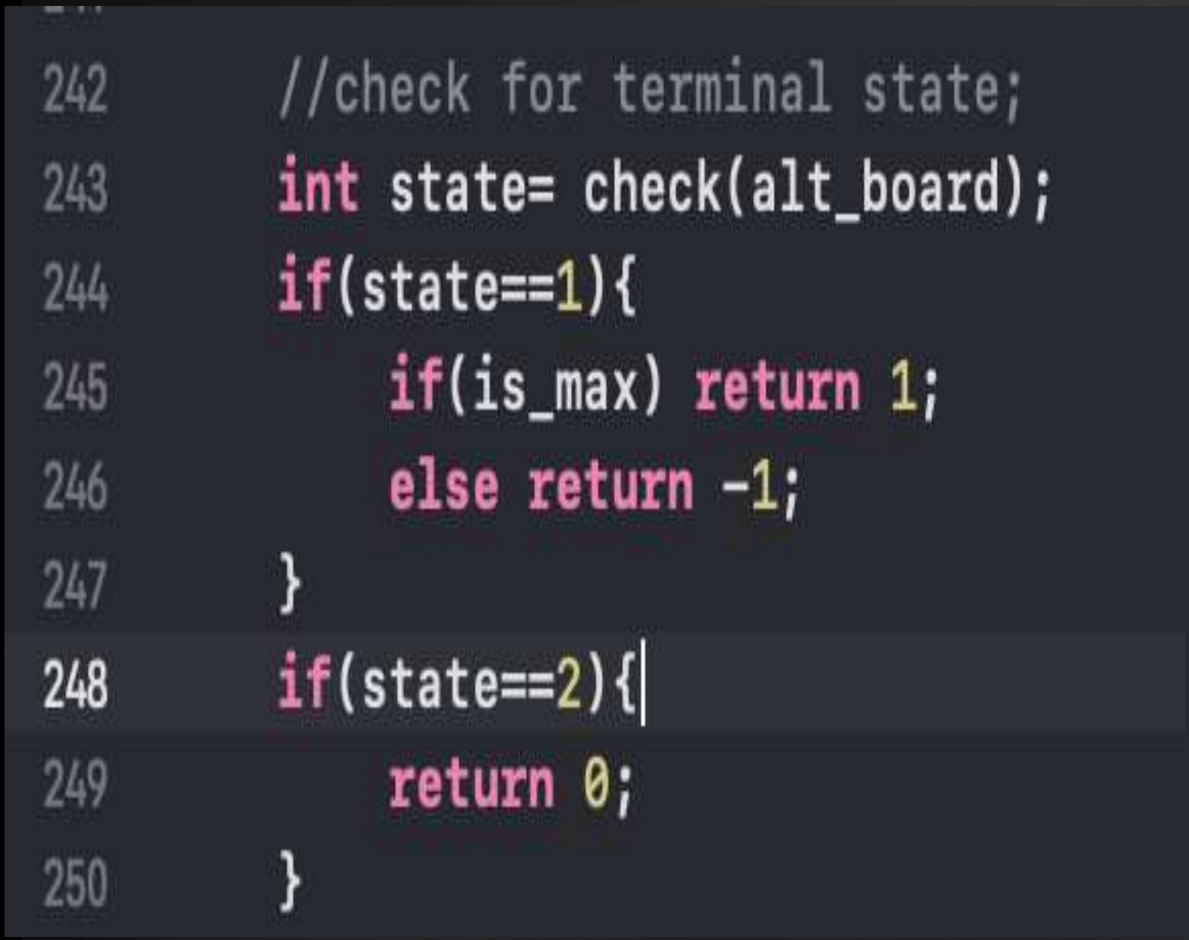
❑ minimax() function



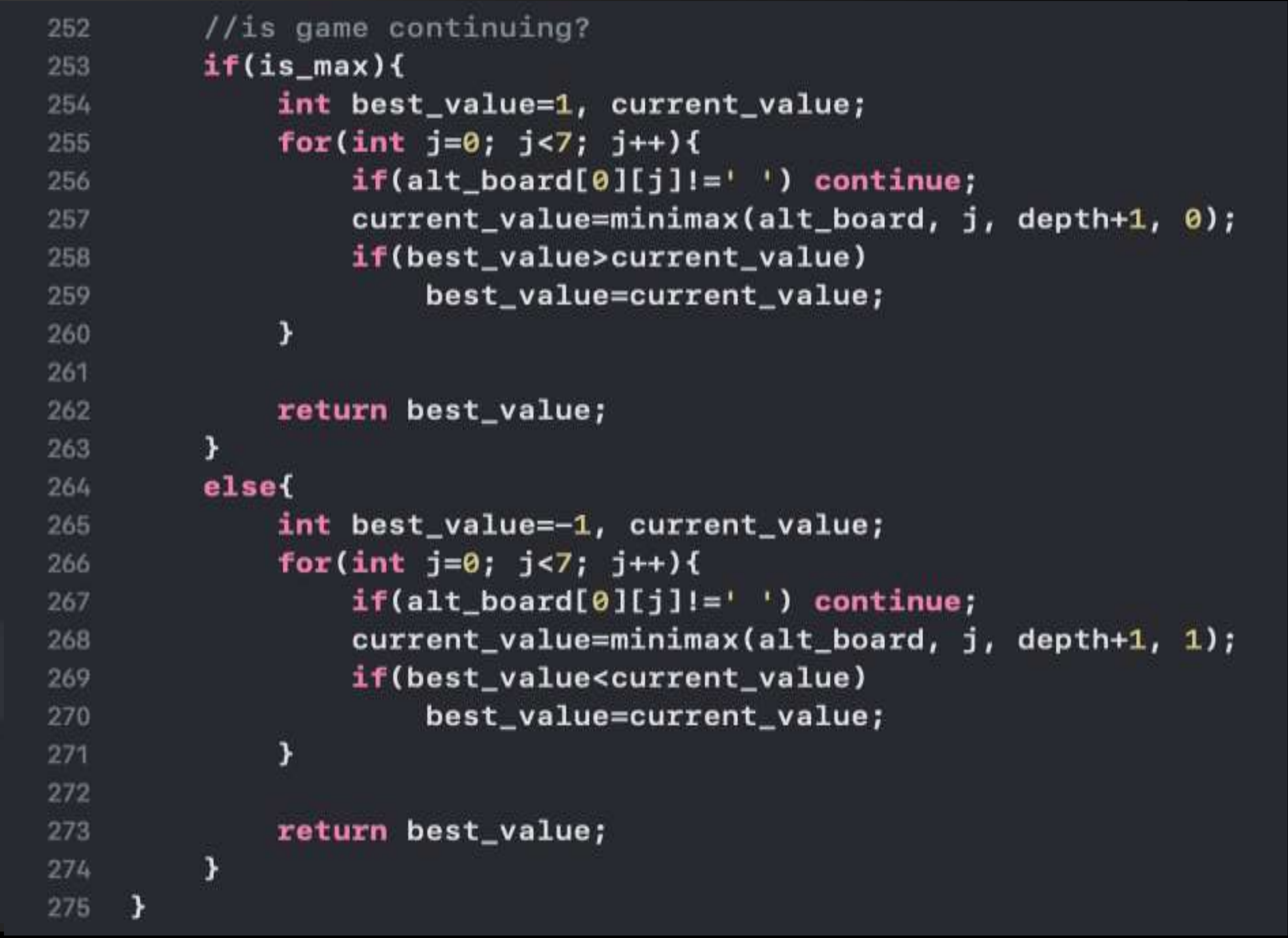
```c
int minimax(char board[6][7], int choice, int depth, int is_max){
    if(depth>max_depth) return -0.5;

    //generate altered board
    int i, j;
    //copy board
    char alt_board[6][7];
    for(i=0; i<6; i++)
        for(j=0; j<7; j++)
            alt_board[i][j]=board[i][j];

    char symbol= is_max?'X':'O';

    //finding i;
    j=choice;
    for(i=1; alt_board[i][j]==' ' && i<6; i++);

    i--;

    alt_board[i][j]=symbol;
```

❑ BASE CASE

❑ INDUCTIVE STEP



```
//check for terminal state;
int state= check(alt_board);
if(state==1){
    if(is_max) return 1;
    else return -1;
}
if(state==2){
    return 0;
}
```



```
        //is game continuing?
        if(is_max){
            int best_value=1, current_value;
            for(int j=0; j<7; j++){
                if(alt_board[0][j]!=' ') continue;
                current_value=minimax(alt_board, j, depth+1, 0);
                if(best_value>current_value)
                    best_value=current_value;
            }

            return best_value;
        }
        else{
            int best_value=-1, current_value;
            for(int j=0; j<7; j++){
                if(alt_board[0][j]!=' ') continue;
                current_value=minimax(alt_board, j, depth+1, 1);
                if(best_value<current_value)
                    best_value=current_value;
            }

            return best_value;
        }
    }
```

THANK YOU