```
#define _CRT_SECURE_NO_WARNINGS #include <stdio.h> #ifndef ROWS #define ROWS 6 #endif #ifndef COLS #define COLS 7 #endif #define CONNECT_N 4 /* Tokens */ #define EMPTY '.' #define TOKEN_P1 'X' #define TOKEN_P2 'O' /* Player types */ #define HUMAN 1 #define COMPUTER 2 /* Function prototypes */ /* Returns 1 if the given column is full, 0 otherwise */ int isColumnFull(char[][COLS], int, int, int); /* Returns 1 if the board is full (no EMPTY cells), 0 otherwise */ int isBoardFull(char[][COLS], int, int); /* Returns 1 if (row, column) is inside the board boundaries, 0 otherwise */ int isInBounds(int, int, int, int); /* Returns index of row where token will land, or -1 if column is full */ int getFreeRow(char[][COLS], int, int, int); /* Place token in column (0-based). Return row index or -1 if illegal */ int makeMove(char[][COLS], int, int, int, char); /* Check if placing playerChar at (row, column) results in a win */ int checkVictory(char[][COLS], int, int, int, int, char); /* Human player: repeatedly asks until a valid non-full column is chosen (0-based) */ int humanChoose(char[][COLS], int, int); /* Returns 1 if (row, column) is part of a sequence of length 'target' of token, 0 otherwise */ int hasSequenceN(char[][COLS], int, int, int, int, char, int); /* Computer player: chooses column according to priority rules */ int computerChoose(char[][COLS], int, int, char, char); /* Runs the full Connect Four game loop */ void runConnectFour(char[][COLS], int, int, int, int); /* Initialize the board cells to EMPTY */ void initBoard(char[][COLS], int, int); /* Print the current state of the board */ void printBoard(char[][COLS], int, int); /* Ask user for player type (human/computer) for the given player number */ int getPlayerType(int); /* Main function: sets up game and starts the loop */ int main() { char board[ROWS][COLS]; printf("Connect Four (%d rows x %d cols)\n", ROWS, COLS); int p1Type = getPlayerType(1); int p2Type = getPlayerType(2); initBoard(board, ROWS, COLS); printBoard(board, ROWS, COLS); runConnectFour(board, ROWS, COLS, p1Type, p2Type); return 0; } /* Prints the entire board and the column numbers underneath */ void printBoard(char board[][COLS], int rows, int cols) { printf("\n"); for (int r = 0; r < rows; r++) { printf("|"); for (int c = 0; c < cols; c++) { putchar(board[r][c]); printf("|"); } printf("\n"); } for (int c = 1; c <= cols; c++) { printf(" %d", c % 10); } printf("\n\n"); } /* Reads and returns the type of the given player: HUMAN or COMPUTER */ int getPlayerType(int playerNumber) { char ch; while (1) { printf("Choose type for player %d: h - human, c - computer: ", playerNumber); int n = scanf(" %c", &ch); if (n != 1) { printf("Input error. Try again.\n"); while (getchar() != '\n'); /* clear input buffer */ continue; } if (ch == 'h' || ch == 'H') { return HUMAN; } if (ch == 'c' || ch == 'C') { return COMPUTER; } printf("Invalid selection. Enter h or c.\n"); while (getchar() != '\n'); /* clear rest of input */ } } /* Check if a given column is full by scanning all rows in that column */ int isColumnFull(char board[][COLS], int rows, int columns, int column) { (void)columns; /* unused parameter */ for (int i = 0; i <
```

```
rows; i++) { if (board[i][column] == EMPTY) { return 0; /* not full */ } } return 1; /* full */ } /* Check if there is any EMPTY cell on the board */ int isBoardFull(char board[][COLS], int rows, int columns) { for (int i = 0; i < rows; i++) { for (int j = 0; j < columns; j++) { if (board[i][j] == EMPTY) { return 0; /* not full */ } } } return 1; /* full */ } /* Check if given (row, column) indices are inside board boundaries */ int isInBounds(int rows, int columns, int row, int column) { if (row < 0 || row >= rows || column < 0 || column >= columns) { return 0; /* out of bounds */ } return 1; /* in bounds */ } /* Find the lowest free row in the given column, starting from bottom */ int getFreeRow(char board[][COLS], int rows, int columns, int column) { (void)columns; /* unused parameter */ for (int r = rows - 1; r >= 0; r--) { if (board[r][column] == EMPTY) { return r; } } return -1; } /* Place a token in a given column, if possible, and return the row index */ int makeMove(char board[][COLS], int rows, int columns, int column, char playerChar) { if (column < 0 || column >= columns) { return -1; /* illegal column */ } int freeRow = getFreeRow(board, rows, columns, column); if (freeRow == -1) { return -1; /* column full */ } board[freeRow][column] = playerChar; return freeRow; } /* Check if the last move at (row, column) created a sequence of CONNECT_N */ int checkVictory(char board[][COLS], int rows, int columns, int row, int column, char playerChar) { for (int dr = -1; dr <= 1; dr++) { for (int dc = -1; dc <= 1; dc++) { if (dr == 0 && dc == 0) { continue; /* skip no-movement */ } int count = 1; /* Check in the positive direction */ int r = row + dr; int c = column + dc; while (isInBounds(rows, columns, r, c) && board[r][c] == playerChar) { count++; r += dr; c += dc; } /* Check in the negative direction */ r = row - dr; c = column - dc; while (isInBounds(rows, columns, r, c) && board[r][c] == playerChar) { count++; r -= dr; c -= dc; } if (count >= CONNECT_N) { return 1; /* victory */ } } } return 0; /* no victory */ } /* Handle human input, including all error messages and re-prompts */ int humanChoose(char board[][COLS], int rows, int columns) { int column = 0; printf("Enter column (1-%d): ", columns); int scan = scanf("%d", &column); while (scan != 1) { printf("Invalid input. Enter a number.\n"); printf("Enter column (1-%d): ", columns); int ch; while ((ch = getchar()) != '\n' && ch != EOF) { } scan = scanf("%d", &column); } /* Repeat until the user picks a valid, non-full column */ while (!(1 <= column && column <= columns) || isColumnFull(board, rows, columns, column - 1) == 1) { if (!(1 <= column && column <= columns)) { printf("Invalid column. Choose between 1 and %d.\n", columns); printf("Enter column (1-%d): ", columns); } else if (isColumnFull(board, rows, columns, column - 1) == 1) { printf("Column %d is full. Choose another column.\n", column); printf("Enter column (1-%d): ", columns); } scan = scanf("%d", &column); while (scan != 1) { printf("Invalid input. Enter a number.\n"); printf("Enter column (1-%d): ", columns); int ch; while ((ch = getchar()) != '\n' && ch != EOF) { } scan =
```

```
scanf("%d", &column); } } return column - 1; /* convert to 0-based index */ } /* Check for a sequence of 'target' tokens passing through (row, column) */ int hasSequenceN(char board[][COLS], int rows, int columns, int row, int column, char token, int target) { for (int dr = -1; dr <= 1; dr++) { for (int dc = -1; dc <= 1; dc++) { if (dr == 0 && dc == 0) { continue; } int count = 1; int r = row + dr; int c = column + dc; while (isInBounds(rows, columns, r, c) && board[r][c] == token) { count++; r += dr; c += dc; } r = row - dr; c = column - dc; while (isInBounds(rows, columns, r, c) && board[r][c] == token) { count++; r -= dr; c -= dc; } if (count >= target) { return 1; } } } return 0; } /* * Computer move selection: * 1. Winning move * 2. Blocking opponent's win * 3. Creating a sequence of three * 4. Blocking opponent's sequence of three * 5. Fallback: choose column by distance from center and availability */ int computerChoose(char board[][COLS], int rows, int columns, char myToken, char opponentToken) { int order[COLS]; int idx = 0; /* Build column order according to distance from center (and left preference) */ if (columns % 2 == 1) { int center = columns / 2; order[idx++] = center; for (int d = 1; d <= center; d++) { int left = center - d; int right = center + d; if (left >= 0) { order[idx++] = left; } if (right < columns) { order[idx++] = right; } } } else { int centerLeft = columns / 2 - 1; int centerRight = centerLeft + 1; order[idx++] = centerLeft; order[idx++] = centerRight; for (int d = 1;; d++) { int left = centerLeft - d; int right = centerRight + d; if (left < 0 && right >= columns) { break; } if (left >= 0) { order[idx++] = left; } if (right < columns) { order[idx++] = right; } } } /* 1. Try to win immediately */ for (int i = 0; i < columns; i++) { int column = order[i]; if (isColumnFull(board, rows, columns, column)) { continue; } int row = getFreeRow(board, rows, columns, column); char old = board[row][column]; board[row][column] = myToken; int win = checkVictory(board, rows, columns, row, column, myToken); board[row][column] = old; if (win) { return column; } } /* 2. Block opponent's winning move */ for (int i = 0; i < columns; i++) { int column = order[i]; if (isColumnFull(board, rows, columns, column)) { continue; } int row = getFreeRow(board, rows, columns, column); char old = board[row][column]; board[row][column] = opponentToken; int block = checkVictory(board, rows, columns, row, column, opponentToken); board[row][column] = old; if (block) { return column; } } /* 3. Create a sequence of three */ for (int i = 0; i < columns; i++) { int column = order[i]; if (isColumnFull(board, rows, columns, column)) { continue; } int row = getFreeRow(board, rows, columns, column); char old = board[row][column]; board[row][column] = myToken; int good = hasSequenceN(board, rows, columns, row, column, myToken, 3); board[row][column] = old; if (good) { return column; } } /* 4. Block opponent's sequence of three */ for (int i = 0; i < columns; i++) { int column = order[i]; if (isColumnFull(board, rows, columns, column)) { continue; } int row = getFreeRow(board, rows, columns, column); char old = board[row][column];
```

board[row][column] = opponentToken; int good = hasSequenceN(board, rows, columns, row, column, opponentToken, 3); board[row][column] = old; if (good) { return column; } } /* 5. Fallback: choose the first non-full column by ordering rule */ for (int i = 0; i < columns; i++) { int column = order[i]; if (!isColumnFull(board, rows, columns, column)) { return column; } } return -1; /* should not reach here */ } /* Main game loop: alternates turns until win or tie */ void runConnectFour(char board[][COLS], int rows, int columns, int p1Type, int p2Type) { int currentPlayer = 1; int winner = 0; while (!winner && !isBoardFull(board, rows, columns)) { int column; int row; char token = (currentPlayer == 1) ? TOKEN_P1 : TOKEN_P2; int type = (currentPlayer == 1) ? p1Type : p2Type; char oppToken = (currentPlayer == 1) ? TOKEN_P2 : TOKEN_P1; printf("Player %d (%c) turn.\n", currentPlayer, token); if (type == HUMAN) { column = humanChoose(board, rows, columns); } else { column = computerChoose(board, rows, columns, token, oppToken); printf("Computer chose column %d\n", column + 1); } row = makeMove(board, rows, columns, column, token); printBoard(board, rows, columns); if (row >= 0 && checkVictory(board, rows, columns, row, column, token)) { winner = currentPlayer; break; } if (isBoardFull(board, rows, columns)) { break; } currentPlayer = (currentPlayer == 1) ? 2 : 1; } if (winner) { char token = (winner == 1) ? TOKEN_P1 : TOKEN_P2; printf("Player %d (%c) wins!\n", winner, token); } else { printf("Board full and no winner. It's a tie!\n"); } } /* Initialize all cells of the board to EMPTY token */ void initBoard(char board[][COLS], int rows, int columns) { for (int r = 0; r < rows; r++) { for (int c = 0; c < columns; c++) { board[r][c] = EMPTY; } } } make this code python based with this added logic: N-Connect Logic: Since we want to make our board’s size adjustable, we are going to change the length of the sequence we need to mark. Anything less than 2 or greater than 100 (101 and above, rows or columns) - invalid. If 2 is chosen for either rows or columns, sequence is 2 (very short game, but that is the user’s responsibility) If either is 3 - game transforms to tic-tac-toe and the board is 3X3 (Ix Igul), two human players only. You can assume no one is going to mark a taken cell. 4 <= rows or columns <= 5 sequence is 3 6 <= rows or columns <= 10 sequence is 4 11 and above - sequence is 5 Tic-Tac-Toe: Users will need to enter 1 number for the cell chosen. Cells are numbered in this format: |1|2|3| |4|5|6| |7|8|9| Columns will not be numbered Good Luck! make sure to follow the prints EXACTLY, also, the tic-tac toe is printed in the same format as the connnect-4, and notice that when asking to enter rows/columns print Enter number of X, that's it, also have them print in seperate lines

also incase tictactoe is chosen print Tic Tac Toe (Human vs Human) instead of the connect 4 line, and Enter position not enter cell, and in tictactoe dont print Player(X)

turn, and Connect Four - Or More [Or Less] (4 rows x 4 cols, connect 3) Choose type for player 1: h - human, r - random/simple computer, s - strategic computer: Choose type for player 2: h - human, r - random/simple computer, s - strategic these need to print instead of their current counterpart, in the top one, the rows and cols are according to what was chosen, and connect X is according to how much you need to connect to win

generate full code