Adversarial Search: Playing Connect 4

Student Name: Vishakha Kishor Satpute

I have used the following AI tools: [list tools]

I understand that my submission needs to be my own work: VS

Instructions

Total Points: Undegraduates 100, graduate students 110

Complete this notebook and submit it. The notebook needs to be a complete project report with your implementation, documentation including a short discussion of how your implementation works and your design choices, and experimental results (e.g., tables and charts with simulation results) with a short discussion of what they mean. Use the provided notebook cells and insert additional code and markdown cells as needed.

Introduction

You will implement different versions of agents that play Connect 4:

> "Connect 4 is a two-player connection board game, in which the players choose a color and then take turns dropping colored discs into a seven-column, six-row vertically suspended grid. The pieces fall straight down, occupying the lowest available space within the column. The objective of the game is to be the first to form a horizontal, vertical, or diagonal line of four of one's own discs." (see Connect Four on Wikipedia)

Note that Connect-4 has been solved in 1988. A connect-4 solver with a discussion of how to solve different parts of the problem can be found here: https://connect4.gamesolver.org/en/

Task 1: Defining the Search Problem [10 point]

Define the components of the search problem:

- Initial state
- Actions
- Transition model (result function)
- Goal state (terminal state and utility)

Answer:

Initial State:

The initial state of a Connect Four game is an empty game board.

Actions:

An action in Connect 4 corresponds to a legal move that a player can make. In this game, players take turns dropping one of their discs into any one columns. The disc then falls to the lowest available position within that column.

Transition Model (Result Function):

The transition model, or result function, defines how the state of the game changes after a legal move is made. For Connect 4, the result function involves updating the game board to reflect the new position of the dropped disc and changing the player's turn.

Goal State (Terminal State and Utility):

The goal state is a terminal state in which the game is won, lost, or drawn. In Connect 4, a player wins by connecting four of their own discs vertically, horizontally, or diagonally. The utility function assigns values to different terminal states, such as +1 for a win, -1 for a loss, and 0 for a draw.

How big is the state space? Give an estimate and explain it.

Answer: Connect 4 is played on a 7x6 grid. Each cell can be either empty, contain a red disc, or contain a yellow disc. So, each cell has 3 possible states. With 42 cells in total (7 columns x 6 rows), the total number of possible board configurations is 3 raise to 42

How big is the game tree that minimax search will go through? Give an estimate and explain it.

Answer: Branching factor: Considering the board size is 6x7, the branching factor is 7

Depth of the tree: The depth of the tree depends on how many moves can be made before the game reaches a terminal state. In the worst case, if the board is filled without a winner, there will be 42 moves (6 rows X 7 columns).

Size of the game = 7 raise to 42

Task 2: Game Environment and Random Agent [25 point]

Use a numpy character array as the board.

In [2]:
```
import numpy as np

def empty_board(shape=(6, 7)):
    return np.full(shape=shape, fill_value=0)

print(empty_board())
```

```
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]]
```

The standard board is 6×7 but you can use smaller boards to test your code. Instead of colors (red and yellow), I use 1 and -1 to represent the players. Make sure that your agent functions all

have the from: `agent_type(board, player = 1)` , where board is the current board position (in the format above) and player is the player whose next move it is and who the agent should play (as 1 and -1).

In [37]:
```
# Visualization code by Randolph Rankin

import matplotlib.pyplot as plt

def visualize(board):
    plt.axes()
    rectangle=plt.Rectangle((-0.5,len(board)*-1+0.5),len(board[0]),len(board),fc='bl
    circles=[]
    for i,row in enumerate(board):
        for j,val in enumerate(row):
            color='white' if val==0 else 'red' if val==1 else 'yellow'
            circles.append(plt.Circle((j,i*-1),0.4,fc=color))

    plt.gca().add_patch(rectangle)
    for circle in circles:
        plt.gca().add_patch(circle)

    plt.axis('scaled')
    plt.show()
```

Implement helper functions for:

- A check for available actions in each state `actions(s)` .
- The transition model `result(s, a)` .
- Check for terminal states `terminal(s)` .
- The utility function `utility(s)` .

Make sure that all these functions work with boards of different sizes (number of columns and rows).

In [4]:
```
import numpy as np

def initialize_board(rows=6,cols=7):
  return np.zeros((rows, cols))
```

Implement an agent that plays randomly. Make sure the agent function receives as the percept the board and returns a valid action. Use an agent function definition with the following signature (arguments):

```
def random_player(board, player = 1): ...
```

The argument `player` is used for agents that do not store what color they are playing. The value passed on by the environment should be 1 ot -1 for player red and yellow, respectively. See Experiments section for tic-tac-toe for an example.

In [5]:
```
rows,cols = 6,7
def is_valid_action(board, column):
    #Checks if a given column is a valid move.
    return board[rows-1][column] == 0

def update_board(board, row, col, player):
    #Update the board after each action.
```

```
    board[row][col] = player

def get_row(board, col):
    #Returns the row index where a ball can be dropped in the given column.
    for row in range(rows-1, -1, -1):
        if board[row][col] == 0:
            return row
    return -1

def is_terminal_state(board):
    #Checks if the current state is a terminal state (game over).
    return len(valid_actions(board)) == 0 or check_winner(board, 1) or check_winner(

def available_actions(board):
    #Returns a list of available actions (columns) that can be taken in the current
    valid_moves = valid_actions(board)
    return valid_moves

def valid_actions(board):
    #Returns a list of valid moves (columns) where a ball can be dropped.
    valid_moves = []
    for col in range(cols):
        if valid_actions_check(board, col):
            valid_moves.append(col)
    return valid_moves

def valid_actions_check(board, col):
    #Checks if a move (dropping a ball) is valid in the given column.
    return board[0][col] == 0  # If the top row in the column is empty, the move is

def check_winner(board, ball):
  for i in range(cols - 3):
    for j in range(rows):
      if board[j][i] == ball and board[j][i + 1] == ball and board[j][i + 2] == ball
        return True
  for i in range(cols):
    for j in range(rows - 3):
      if board[j][i] == ball and board[j + 1][i] == ball and board[j + 2][i] == ball
        return True
  for i in range(cols - 3):
    for j in range(rows - 3):
      if board[j][i] == ball and board[j + 1][i + 1] == ball and board[j + 2][i + 2]
        return True
  for i in range(cols - 3):
    for j in range(3, rows):
      if board[j][i] == ball and board[j - 1][i + 1] == ball and board[j - 2][i + 2]
        return True
#board = initialize_board(rows,cols)
#visualize(board)
def utility(board, randomAIPlayers = False):
    winner = terminalState(board)
    if winner == 1:
        if randomAIPlayers:
            print("Player 1 is the winner!!!")
        return True
    elif winner == -1:
        if randomAIPlayers:
            print("Player 2 is the winner!!!")
        return True
    return False

def make_move(board, col, player):
    row = get_row(board, col)
    update_board(board, row, col, player)
```

```
def terminalState(board):
    for i in range(rows):
        for j in range(cols):
            if(board[i][j]!=0):
                for a in [(0,1),(1,0),(1,1),(1,-1)]:
                    if check_winner((i,j), a, board, 1):
                        return 1
                    elif check_winner((i,j), a, board, -1):
                        return -1
    return None
```

Let two random agents play against each other 1000 times. Look at the Experiments section for tic-tac-toe to see how the environment uses the agent functions to play against each other.

How often does each player win? Is the result expected?

In [6]:

```
def random_agent(board):
    return np.random.choice(valid_actions(board))

def play_game():
    board = initialize_board(6, 7)
    current_player = 1

    while not is_terminal_state(board):
        action = random_agent(board)
        row = get_row(board, action)
        update_board(board, row, action, current_player)
        current_player = 3 - current_player  # Switch player (1 <-> 2)

    return check_winner(board, 1), check_winner(board, 2)

def main():
    player1_wins, player2_wins = 0, 0

    for _ in range(1000):
        result_player1, result_player2 = play_game()

        if result_player1:
            player1_wins += 1
        elif result_player2:
            player2_wins += 1

    print(f"Player 1 wins: {player1_wins}")
    print(f"Player 2 wins: {player2_wins}")

if __name__ == "__main__":
    main()
```

```
Player 1 wins: 570
Player 2 wins: 428
```

Task 3: Minimax Search with Alpha-Beta Pruning

Implement the Search [20 points]

Implement minimax search starting from a given board for specifying the player. You can use code from the tic-tac-toe example.

Important Notes:

- Make sure that all your agent functions have a signature consistent with the random agent above and that it uses a class to store state information. This is essential to be able play against agents from other students later.
- The search space for a 6×7 board is large. You can experiment with smaller boards (the smallest is 4×4) and/or changing the winning rule to connect 3 instead of 4.

In [51]:

```
import random
import math


#This function evaluates the given game board's state for a specific player (1 or 2)
#It assigns a score (p) based on the player's potential for winning.
#The scoring is based on the presence of player's tokens in rows, columns, and diago
#The function gives higher priority to certain positions, such as the center column.
def evaluate(board, player):
    p = 0

    # Check the center column for potential moves
    center_column = [int(board[i, cols // 2]) for i in range(rows)]
    center_priority = center_column.count(player)
    p += center_priority * 3

    # Check each row for potential moves
    for i in range(rows):
        row = [int(board[i, j]) for j in range(cols)]
        for k in range(cols - 3):
            group = row[k:k + 4]
            if group.count(player) == 4:
                p += 100
            elif group.count(player) == 3 and group.count(0) == 1:
                p += 5
            elif group.count(player) == 2 and group.count(0) == 2:
                p += 2
            if group.count(1) == 3 and group.count(0) == 1:
                p -= 4

    # Check diagonals starting from the top-left
    for i in range(rows - 3):
        for j in range(cols - 3):
            group = [board[i + k][j + k] for k in range(4)]
            if group.count(player) == 4:
                p += 100
            elif group.count(player) == 3 and group.count(0) == 1:
                p += 5
            elif group.count(player) == 2 and group.count(0) == 2:
                p += 2
            if group.count(1) == 3 and group.count(0) == 1:
                p -= 4

    # Check diagonals starting from the top-right
    for i in range(rows - 3):
        for j in range(cols - 3):
            group = [board[i + 3 - k][j + k] for k in range(4)]
            if group.count(player) == 4:
                p += 100
            elif group.count(player) == 3 and group.count(0) == 1:
                p += 5
            elif group.count(player) == 2 and group.count(0) == 2:
                p += 2
            if group.count(1) == 3 and group.count(0) == 1:
                p -= 4
```

```
    return p

#This function implements the Minimax algorithm with alpha-beta pruning for a Connec
#It recursively explores the game tree to find the best move for the AI player (maxp
def minmax(board,maxplayer,a,b,depth):
  terminal=is_terminal_state(board)
  if depth==0 or terminal:
    if terminal:
      if check_winner(board,2):
        return (1,None)
      elif check_winner(board,1):
        return (-1,None)
      else:
        return (0,None)
    elif depth==0:
      return (evaluate(board,2),None)
  elif maxplayer:
    actions=valid_actions(board)
    #print(actions)
    p=-math.inf
    col=random.choice(actions)
    for position_col in actions:
      new_board = board.copy()
      position_row=get_row(new_board,position_col)
      update_board(new_board,position_row,position_col,2) #check by dropping AI ball
      updated_prior=minmax(new_board,False,a,b,depth-1)[0] #0th index represnets the
      if(p<updated_prior):
        p=updated_prior
        col=position_col
      a=max(a,p)
      if a>=b:
        break

    return p,col
  else:
    actions = valid_actions(board)
    col=random.choice(actions)
    p = math.inf
    for position_col in actions:
      new_board = board.copy()
      position_row = get_row(new_board, position_col)
      update_board(new_board, position_row, position_col, 1)
      updated_prior = minmax(new_board, True,a,b, depth - 1)[0]
      if (p>updated_prior):
        col = position_col
        priority=updated_prior
      b = min(b, p)
      if a >= b:
        break
    return p,col

def minmax(board,maxplayer,a,b,depth):
  terminal=is_terminal_state(board)
  if depth==0 or terminal:
    if terminal:
      if check_winner(board,2):
        return (1,None)
      elif check_winner(board,1):
        return (-1,None)
      else:
        return (0,None)
    elif depth==0:
      return (evaluate(board,2),None)
```

```
  elif maxplayer:
    actions=valid_actions(board)
    #print(actions)
    p=-math.inf
    col=random.choice(actions)
    for position_col in actions:
      new_board = board.copy()
      position_row=get_row(new_board,position_col)
      update_board(new_board,position_row,position_col,2) #check by dropping AI ball
      updated_prior=minmax(new_board,False,a,b,depth-1)[0] #0th index represnets the
      if(p<updated_prior):
        p=updated_prior
        col=position_col
      a=max(a,p)
      if a>=b:
        break

    return p,col
  else:
    actions = valid_actions(board)
    col=random.choice(actions)
    p = math.inf
    for position_col in actions:
      new_board = board.copy()
      position_row = get_row(new_board, position_col)
      update_board(new_board, position_row, position_col, 1)
      updated_prior = minmax(new_board, True,a,b, depth - 1)[0]
      if (p>updated_prior):
        col = position_col
        priority=updated_prior
      b = min(b, p)
      if a >= b:
        break
    return p,col
```

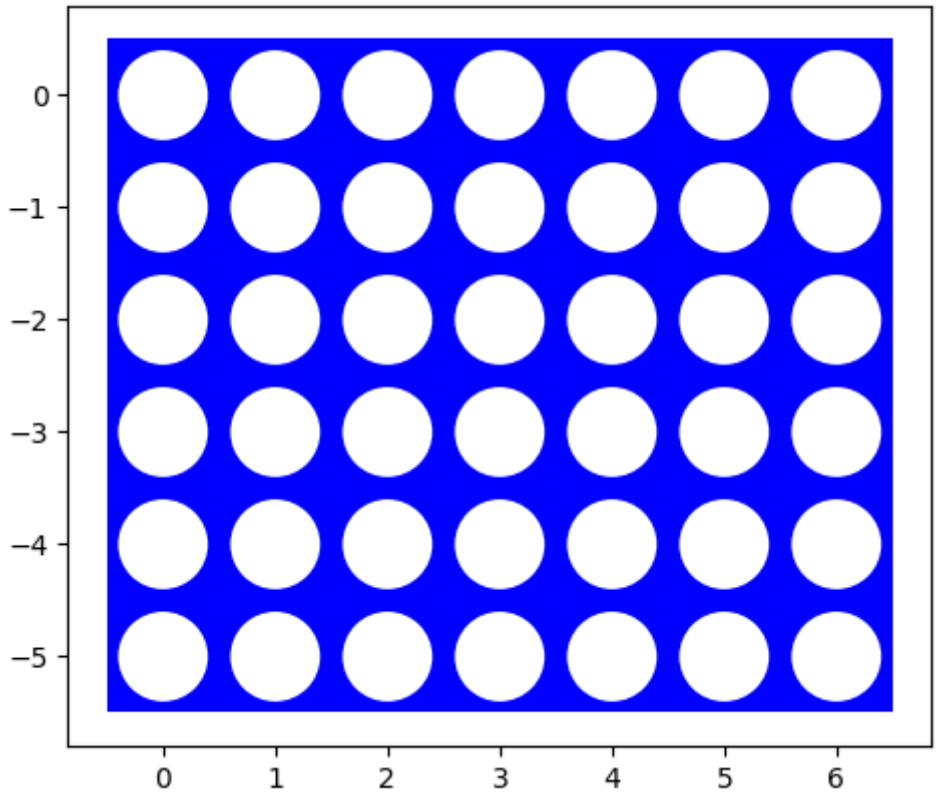
Experiment with some manually created boards (at least 5) to check if the agent spots winning opportunities.

Description: The game is initialized with variables for tracking game state, turn (0 for human, 1 for AI), and difficulty level. The game board is created and visualized using the initialize_board and visualize functions. The main game loop (while not game_over_check) allows players to take turns until the game is over. For the human player's turn (turn == 0), the player is prompted to input a column, and the move is made if the column is valid. The board is then updated and visualized. If the human player wins, the game ends. For the AI player's turn (turn == 1), the Minimax algorithm with alpha-beta pruning (minmax function) is used to determine the best move. The AI's move is then made, and the board is updated and visualized. If the AI wins, the game ends. The game loop continues until a player wins or the board is filled, resulting in a draw. The game outcome is displayed accordingly.

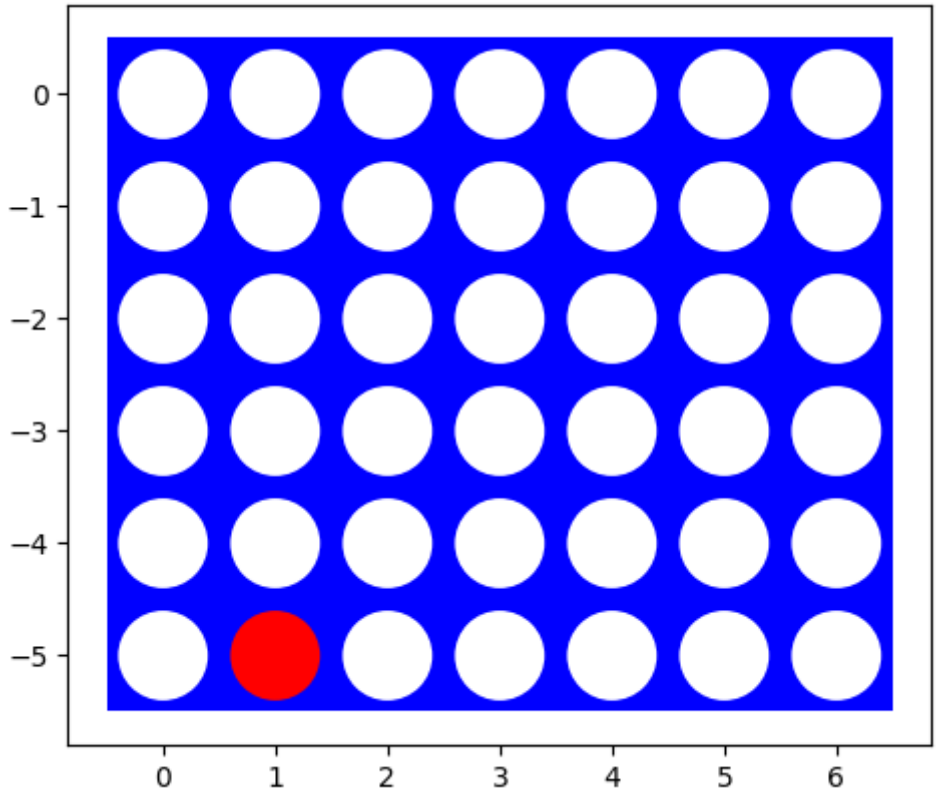
In [45]:

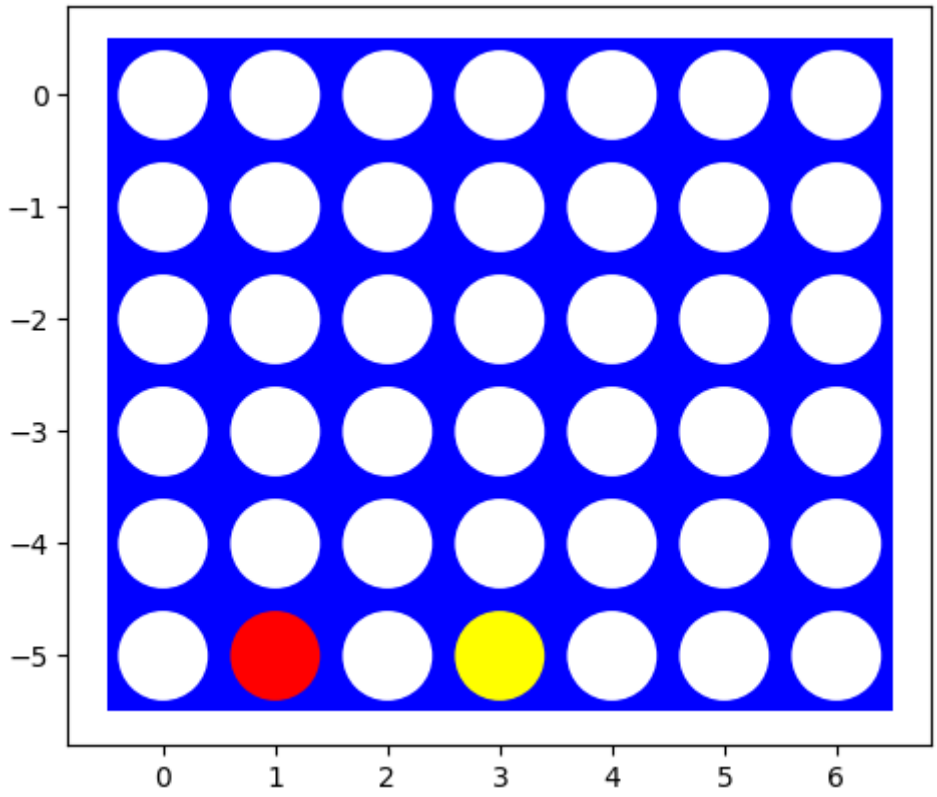
```
game_over_check = False
turn = 0  # Human turn is first
difficulty = 0


board = initialize_board(6,7)
visualize(board)

while not game_over_check:
    if turn == 0:
        col = int(input("Enter the column:"))
```

```
        if valid_actions_check(board, col):
            row = get_row(board, col)
            update_board(board, row, col, 1)
            #print_board(board)
            if check_winner(board,1):
                print("YOU WIN !")
                game_over_check = True
            turn += 1
            visualize(board)
    elif turn == 1:
        func_score, best_col = minmax(board, True, -math.inf, math.inf, 3)
        if valid_actions_check(board, best_col):
            row = get_row(board, best_col)
            update_board(board, row, best_col, 2)
            #print_board(board)
            if check_winner(board, 2):
                print("AI WIN!")
                game_over_check = True
            turn -= 1
            visualize(board)
```

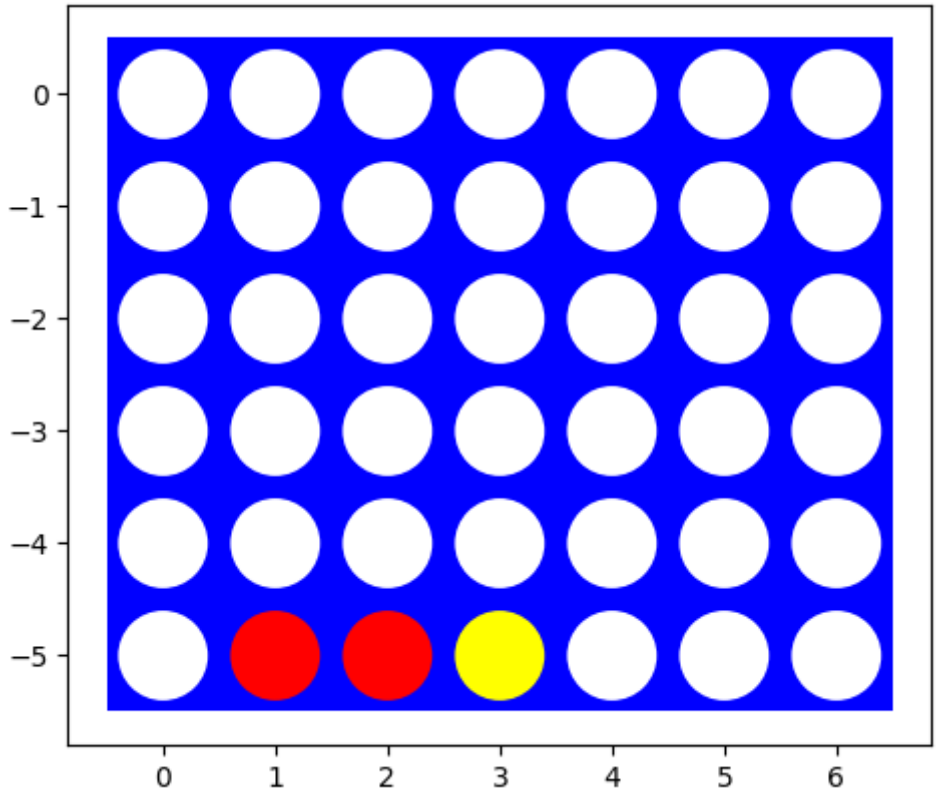


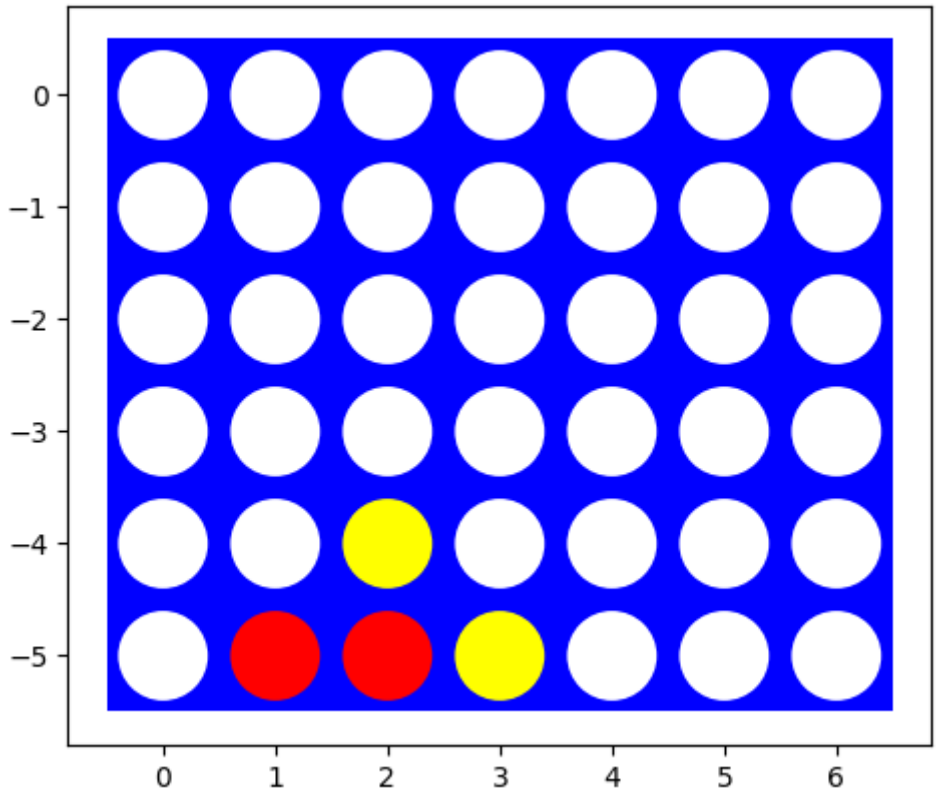

```
Enter the column:1
```



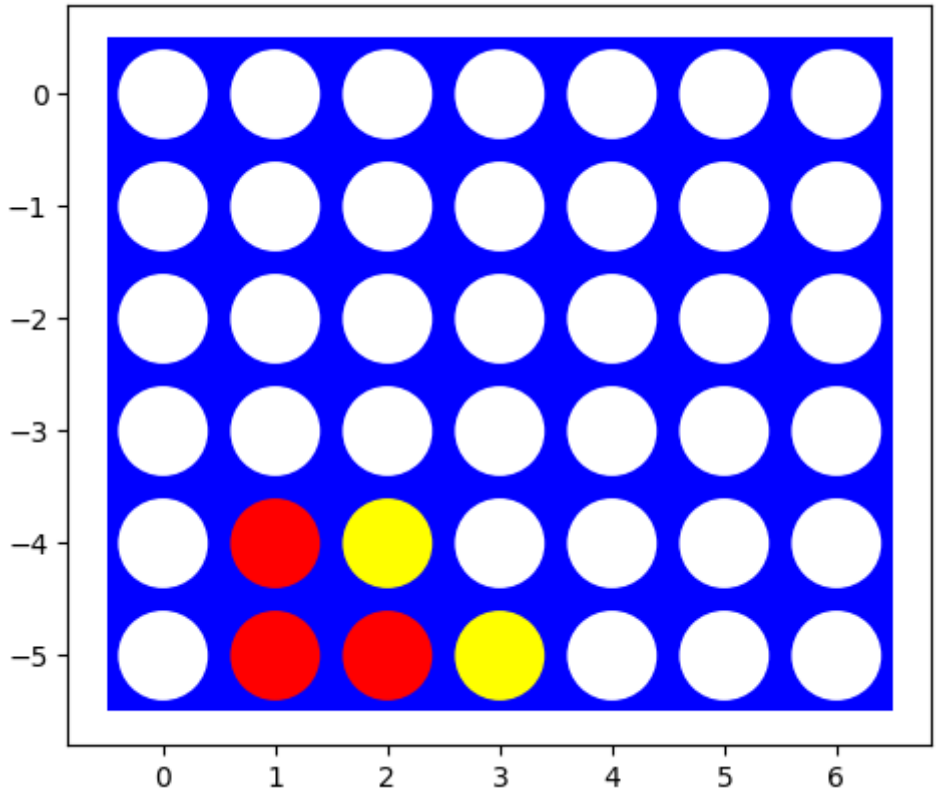


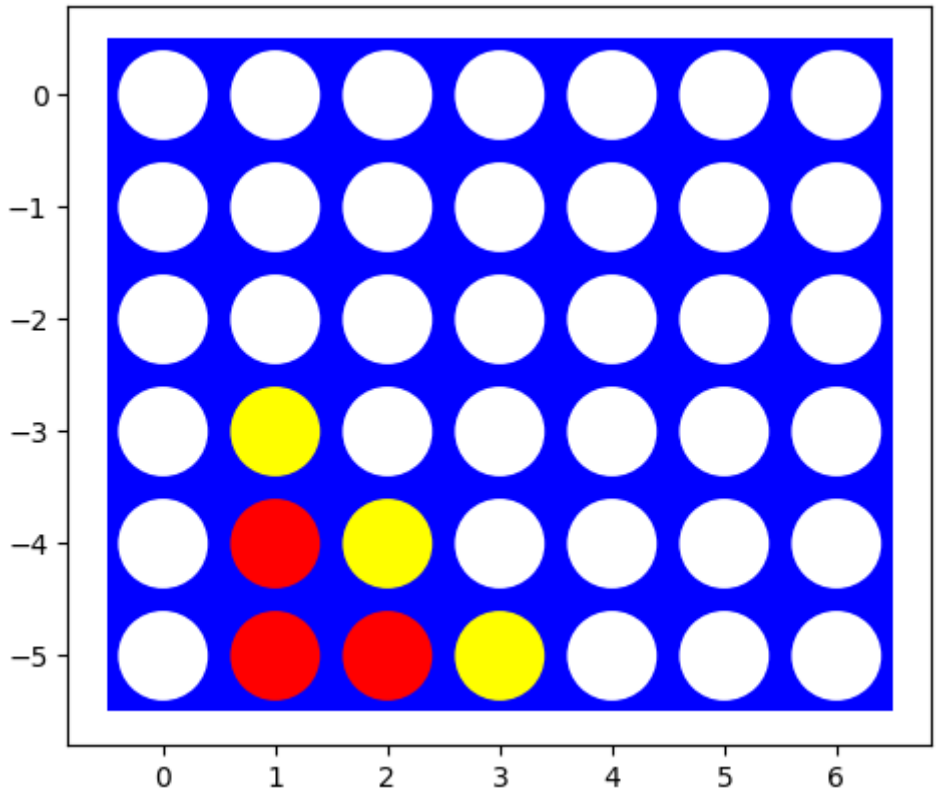

```
Enter the column:2
```



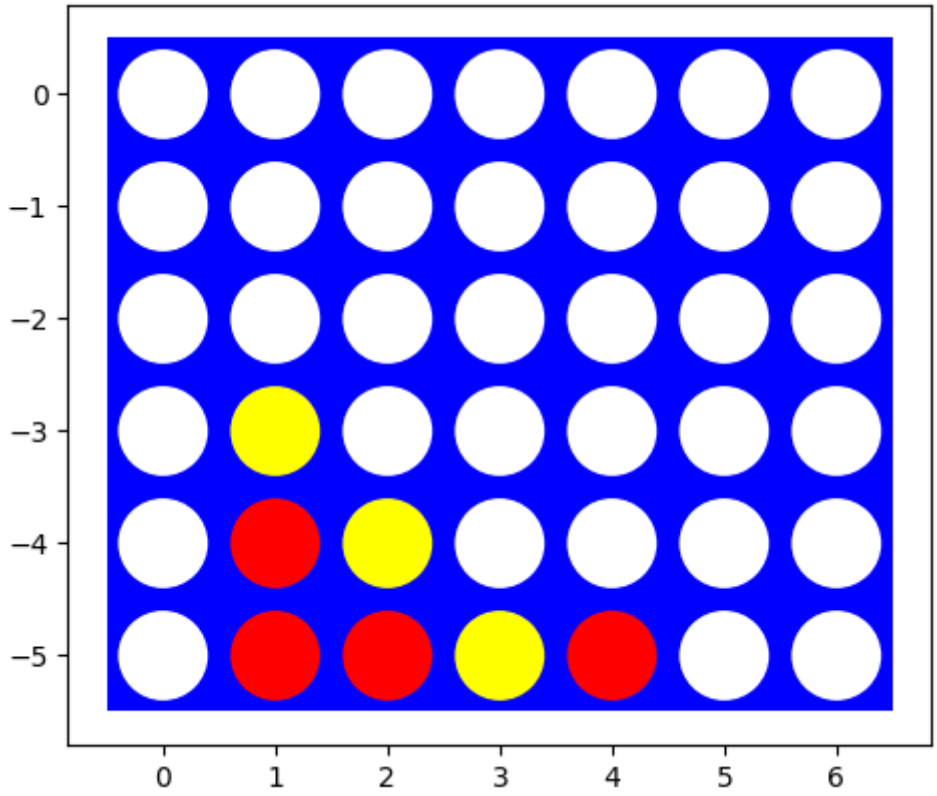


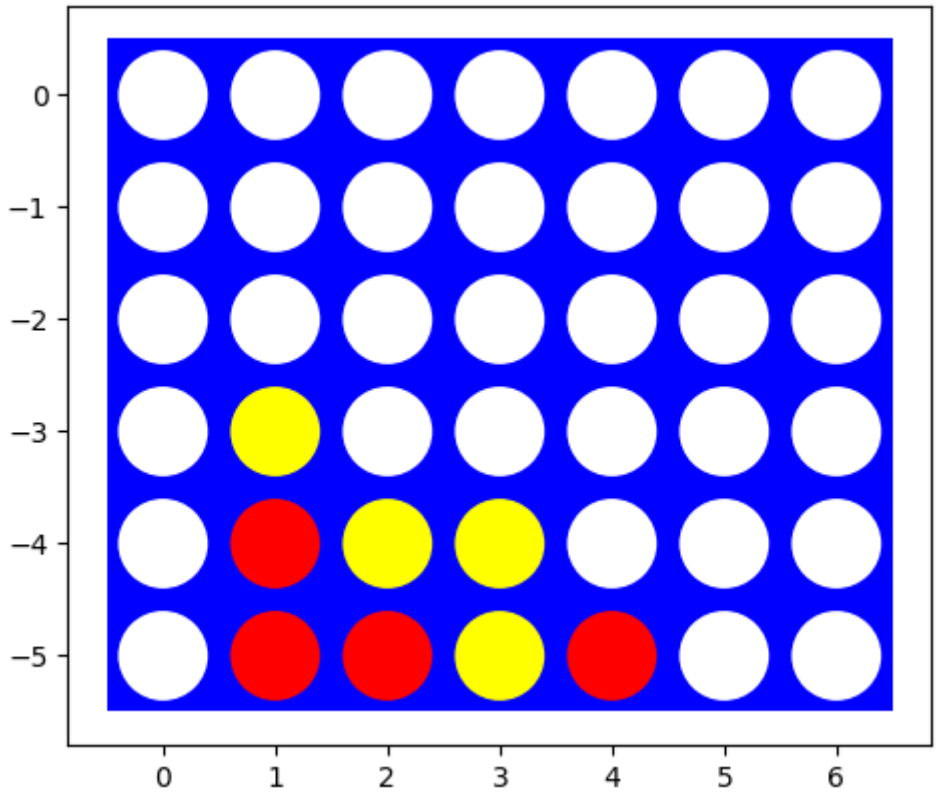

```
Enter the column:1
```



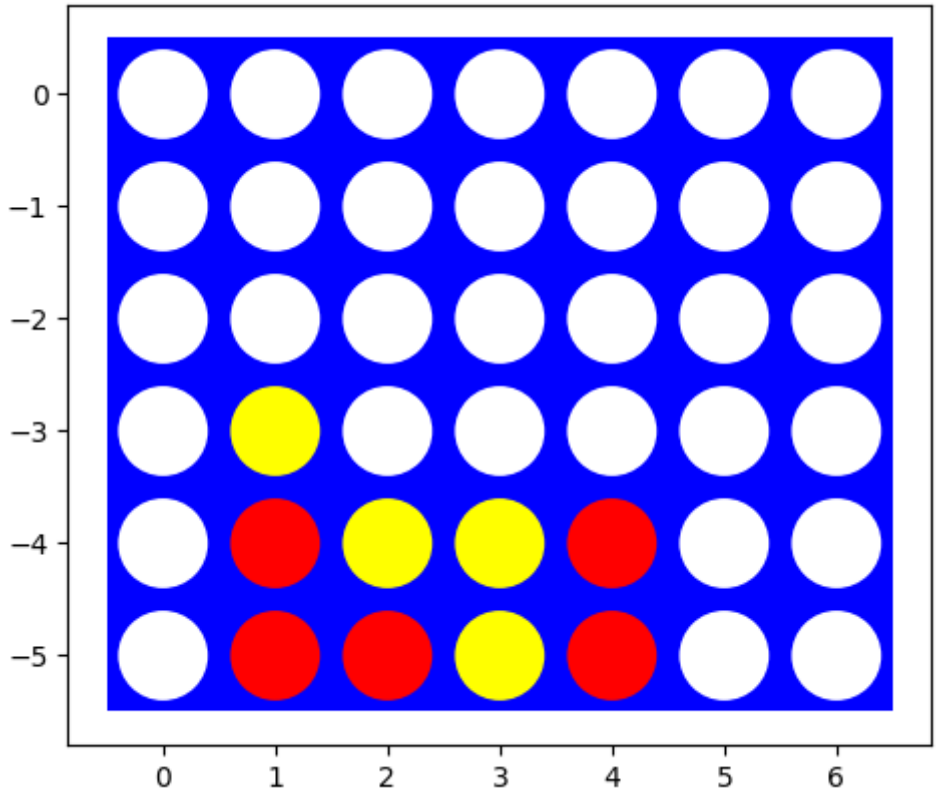


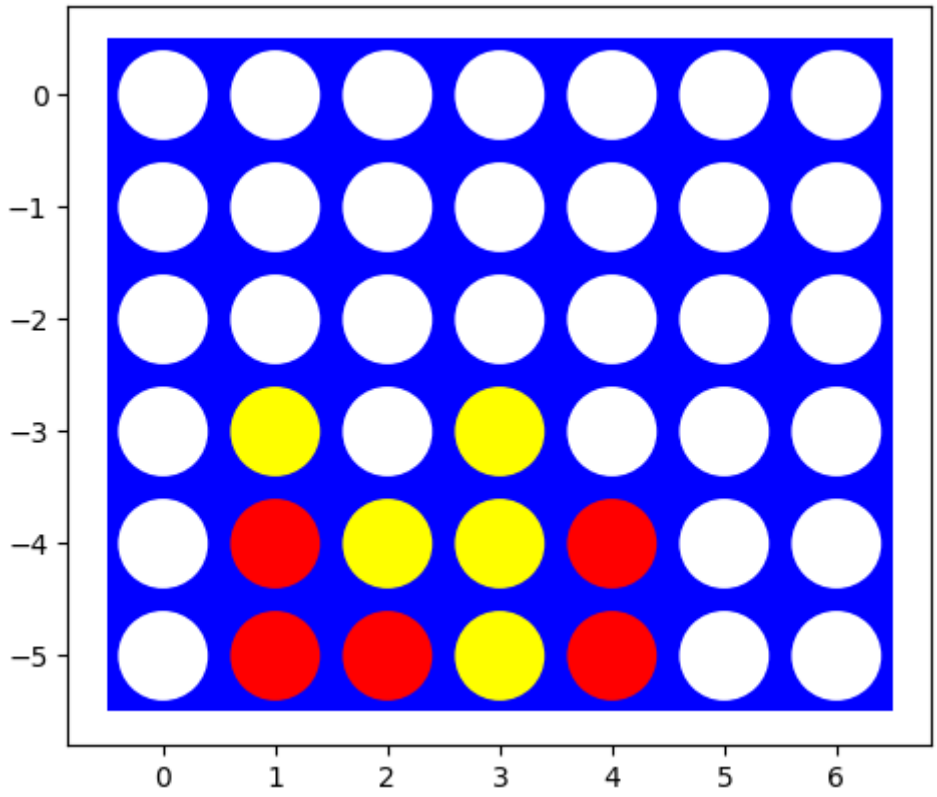

```
Enter the column:4
```



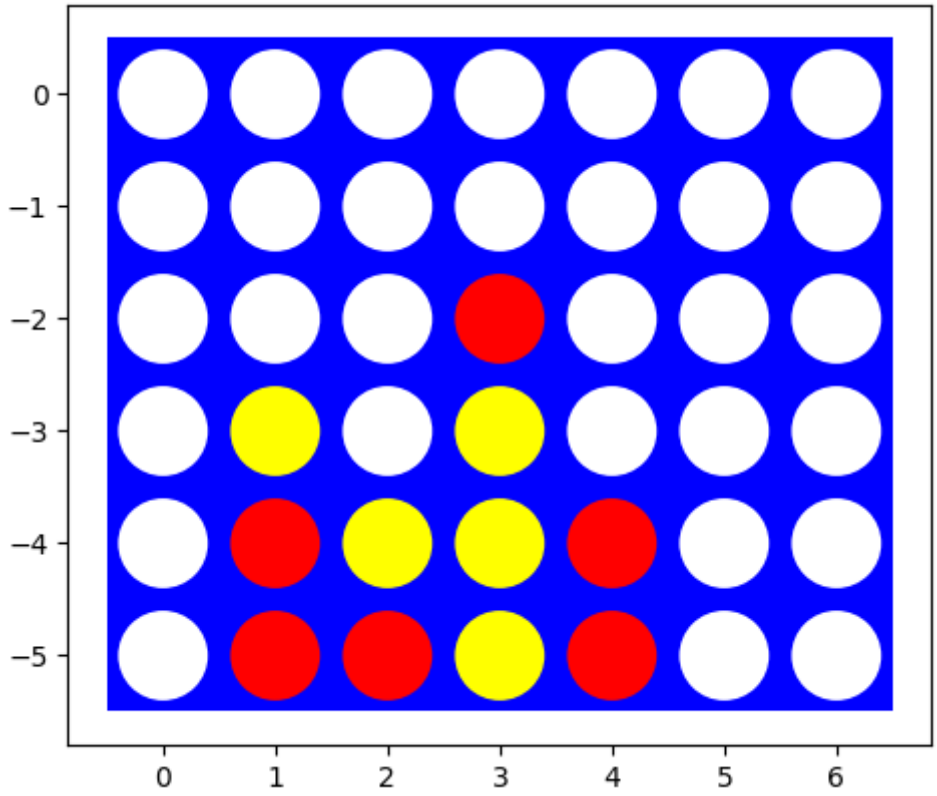


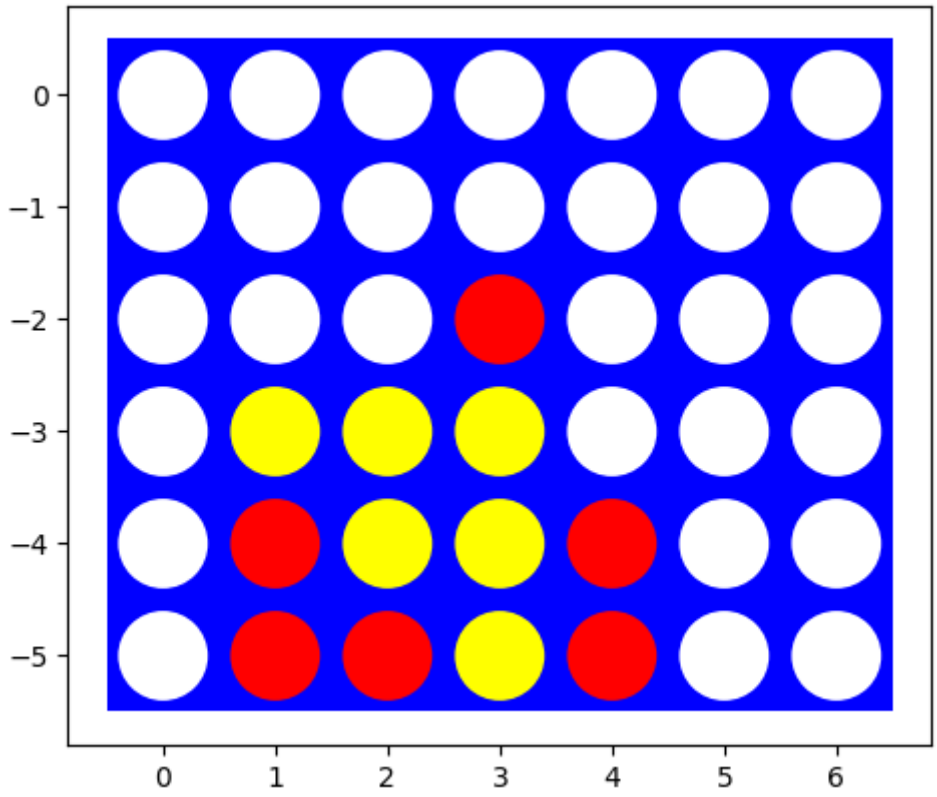

```
Enter the column:4
```



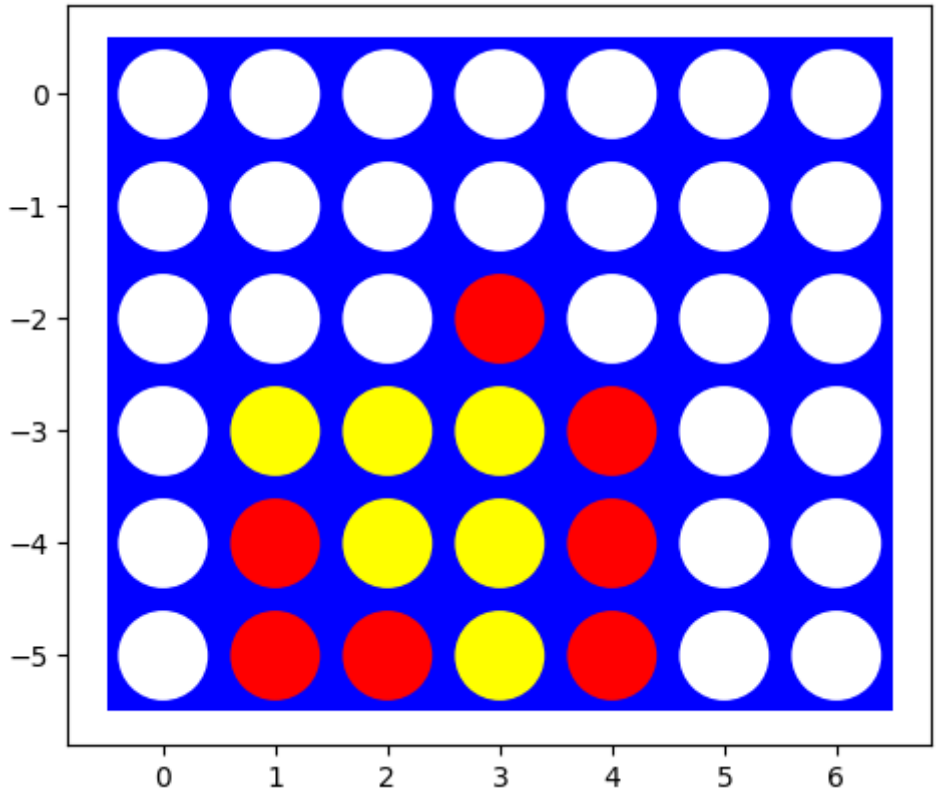


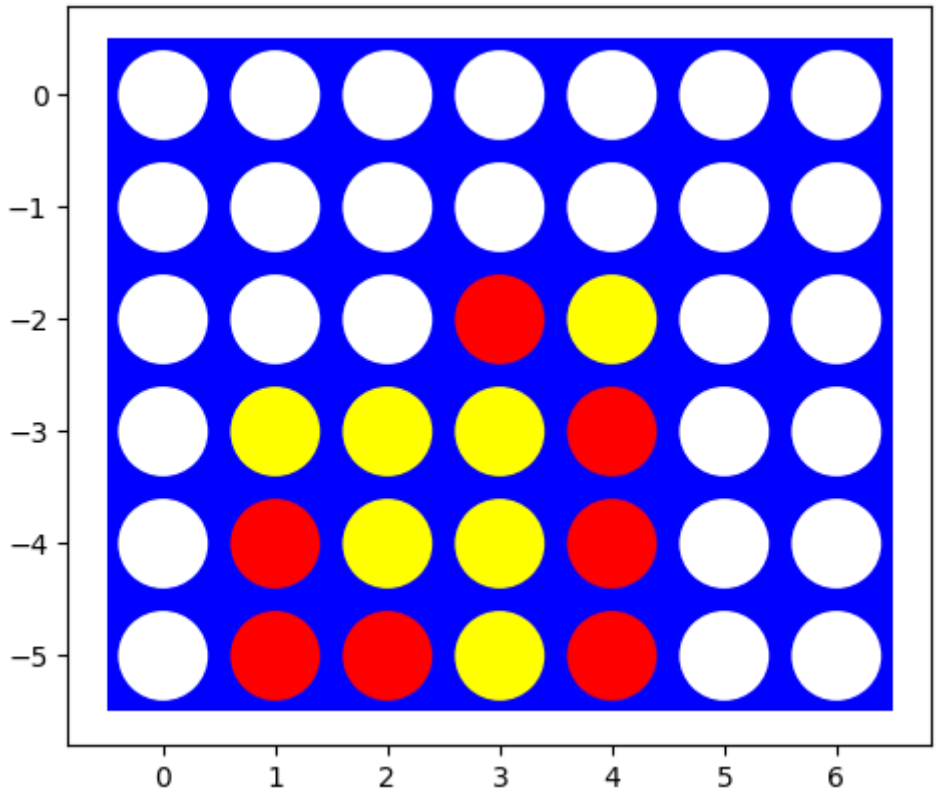

```
Enter the column:3
```



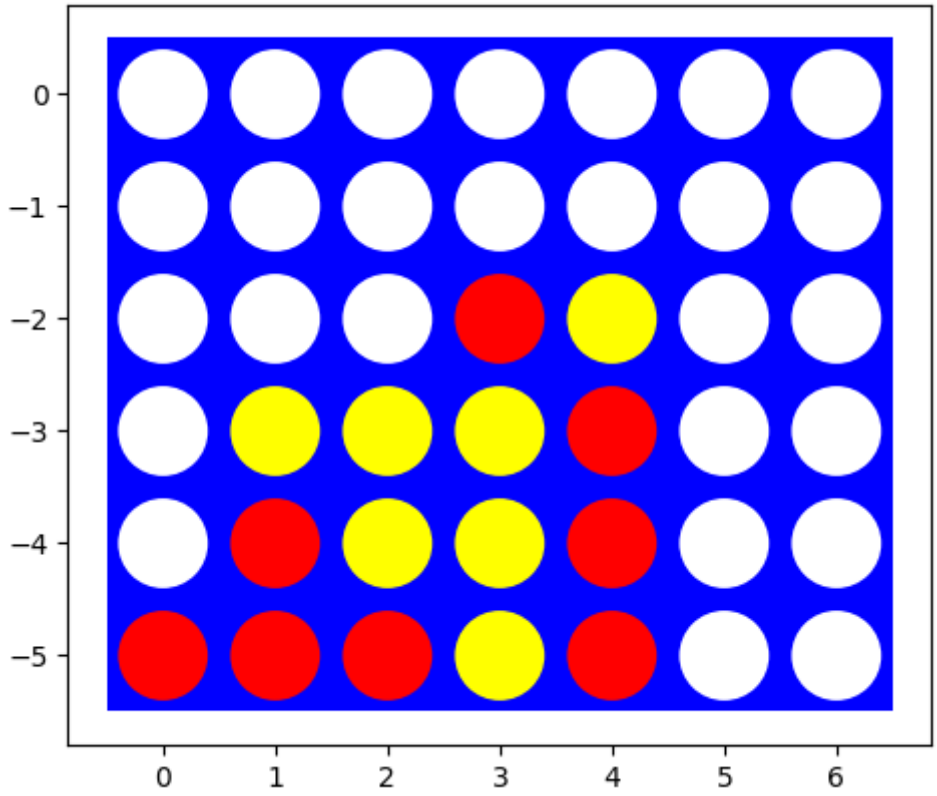


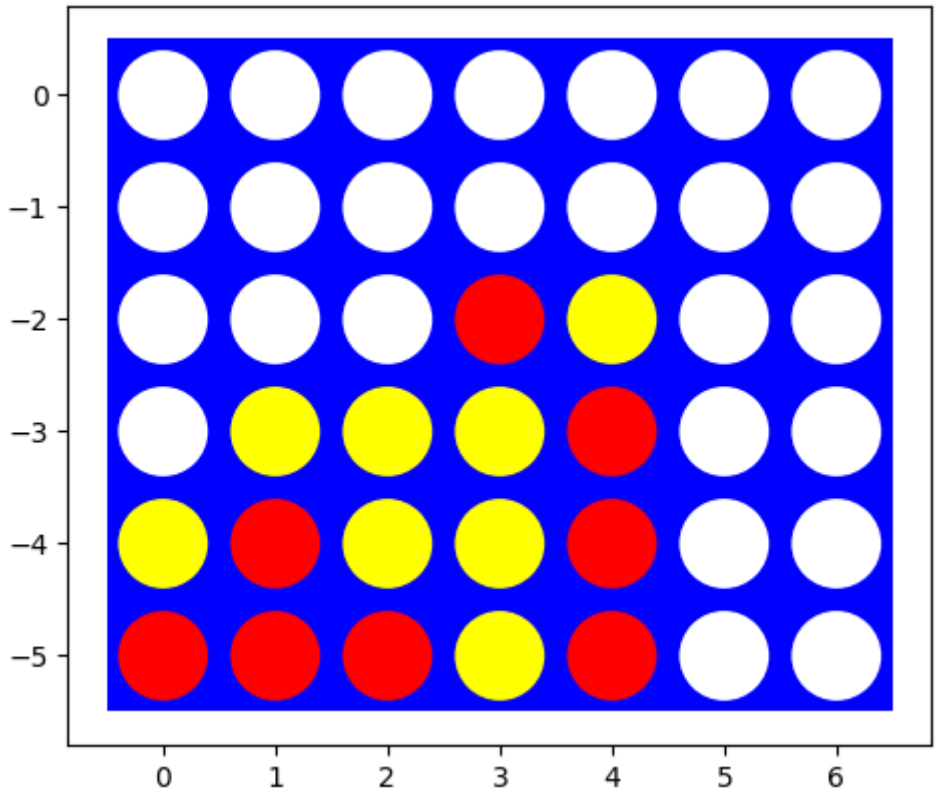

```
Enter the column:4
```



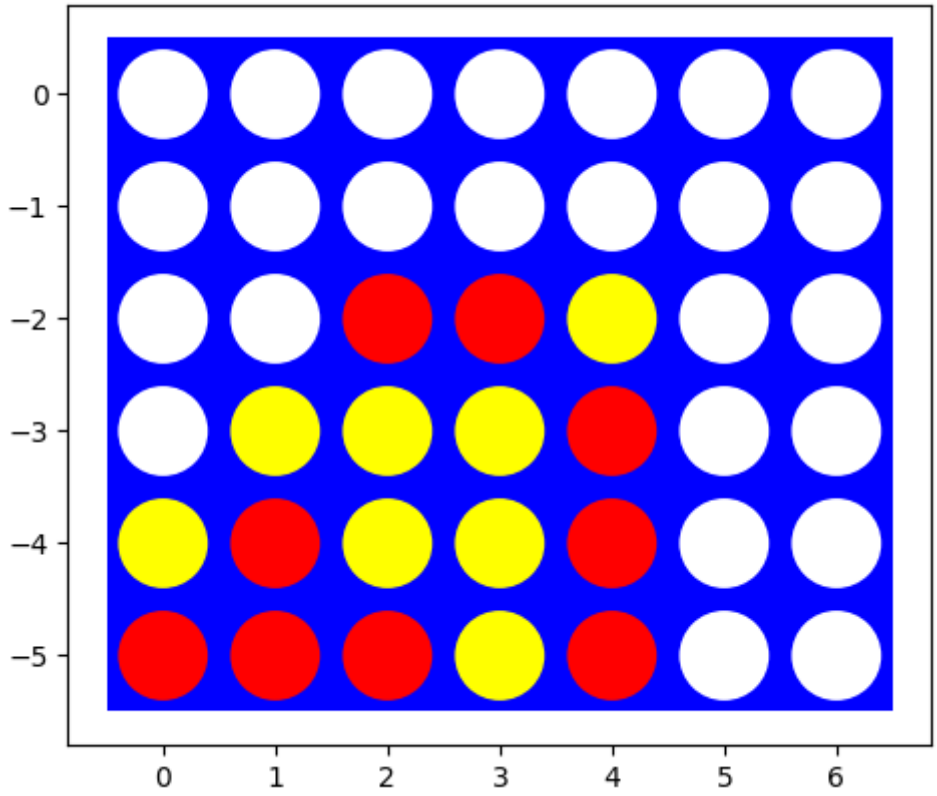



```
Enter the column:0
```

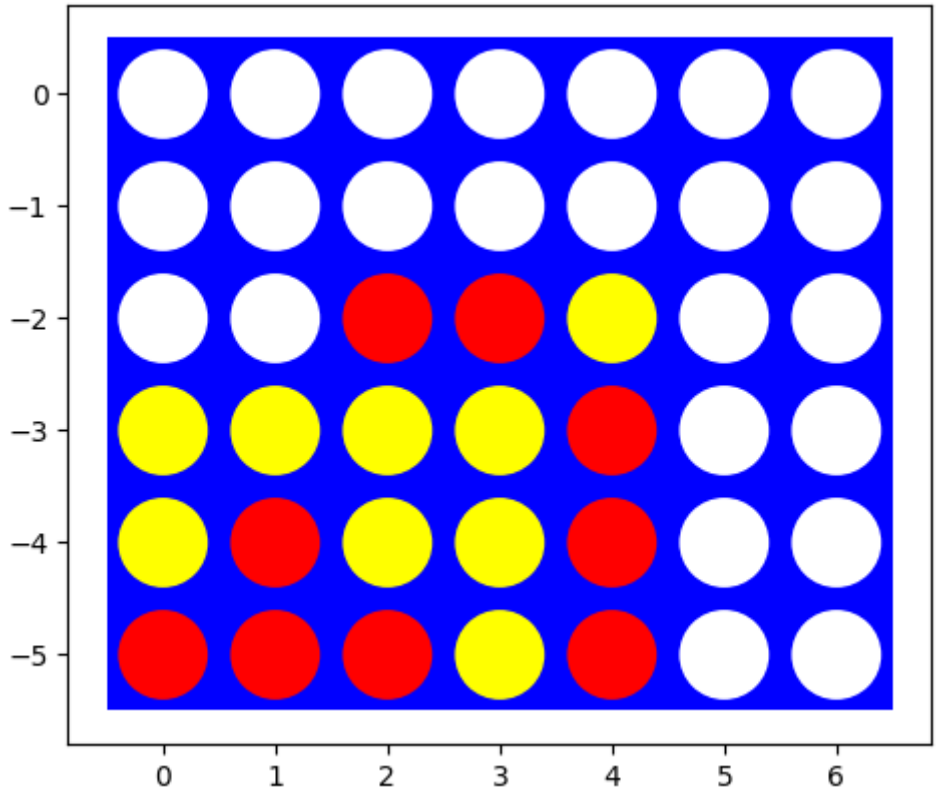





```
Enter the column:2
```



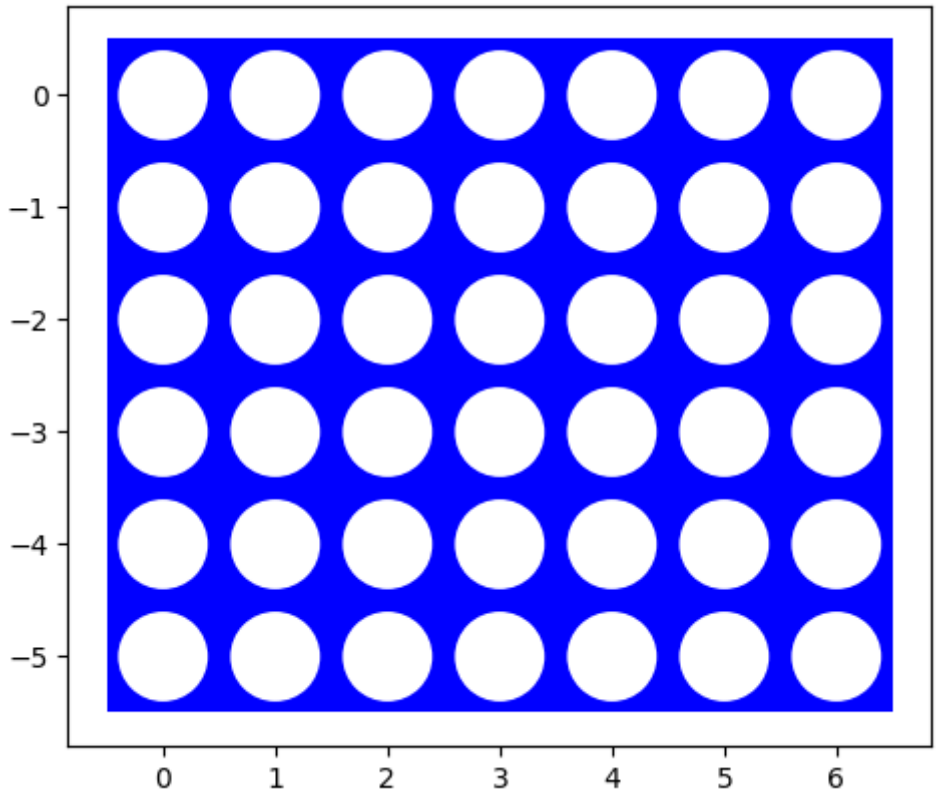
AI WIN!



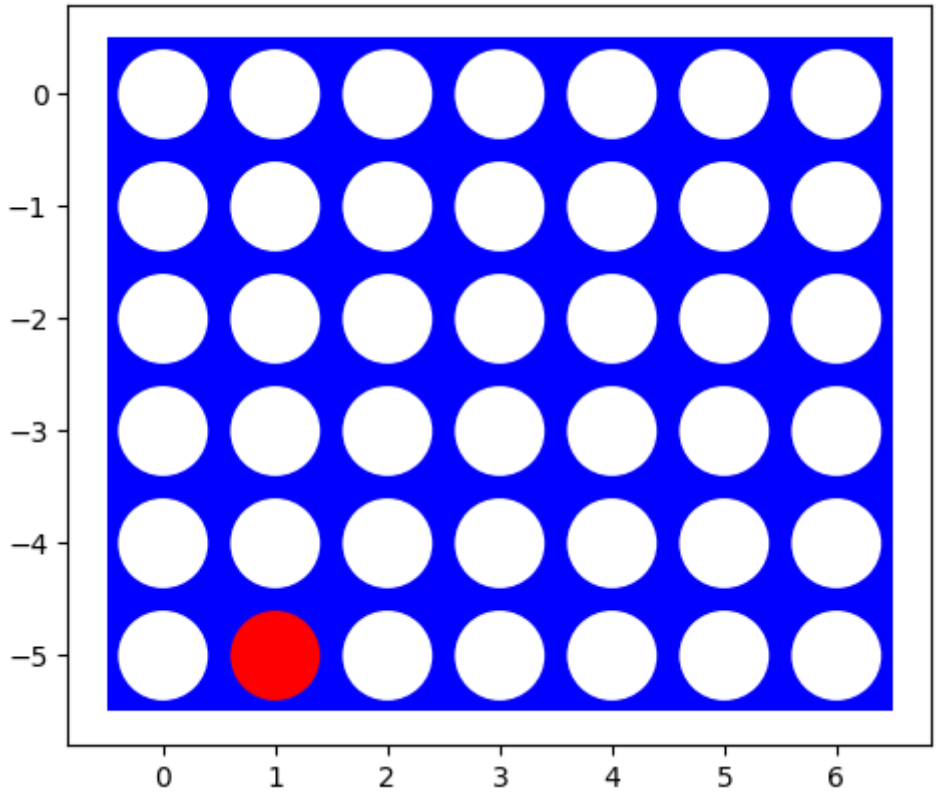
In [49]:

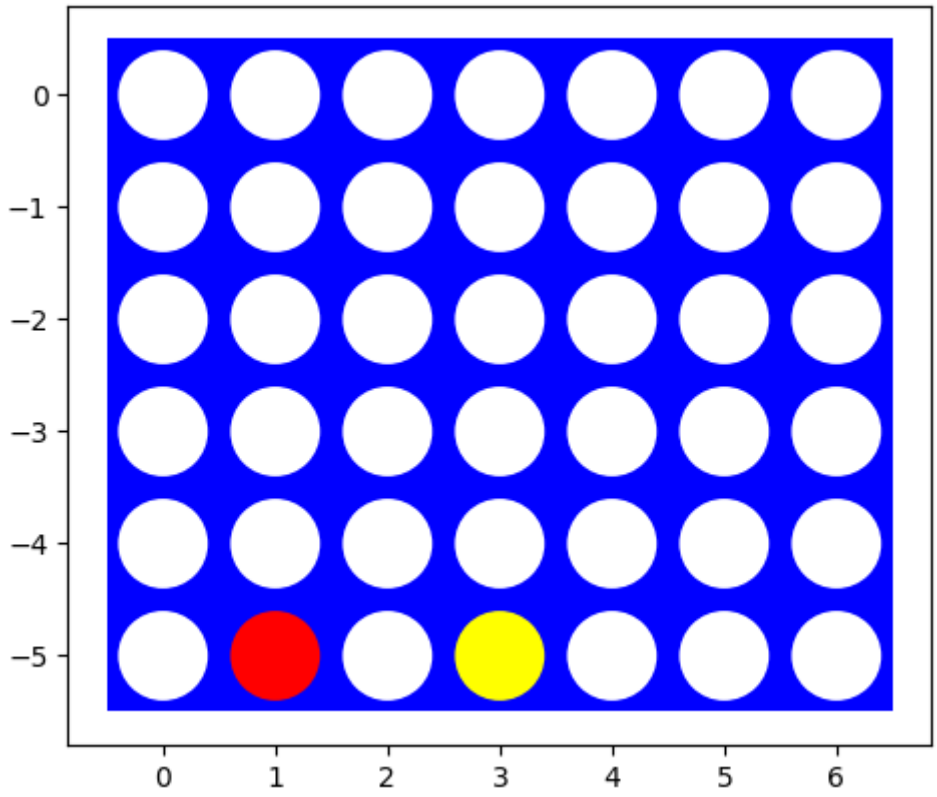
```
game_over_check = False
turn = 0  # Human turn is first
difficulty = 0


board = initialize_board(6,7)
visualize(board)

while not game_over_check:
    if turn == 0:
```

```
        col = int(input("Enter the column:"))
        if valid_actions_check(board, col):
            row = get_row(board, col)
            update_board(board, row, col, 1)
            #print_board(board)
            if check_winner(board,1):
                print("YOU WIN !")
                game_over_check = True
            turn += 1
            visualize(board)
    elif turn == 1:
        func_score, best_col = minmax(board, True, -math.inf, math.inf, 3)
        if valid_actions_check(board, best_col):
            row = get_row(board, best_col)
            update_board(board, row, best_col, 2)
            #print_board(board)
            if check_winner(board, 2):
                print("AI WIN!")
                game_over_check = True
            turn -= 1
            visualize(board)
```

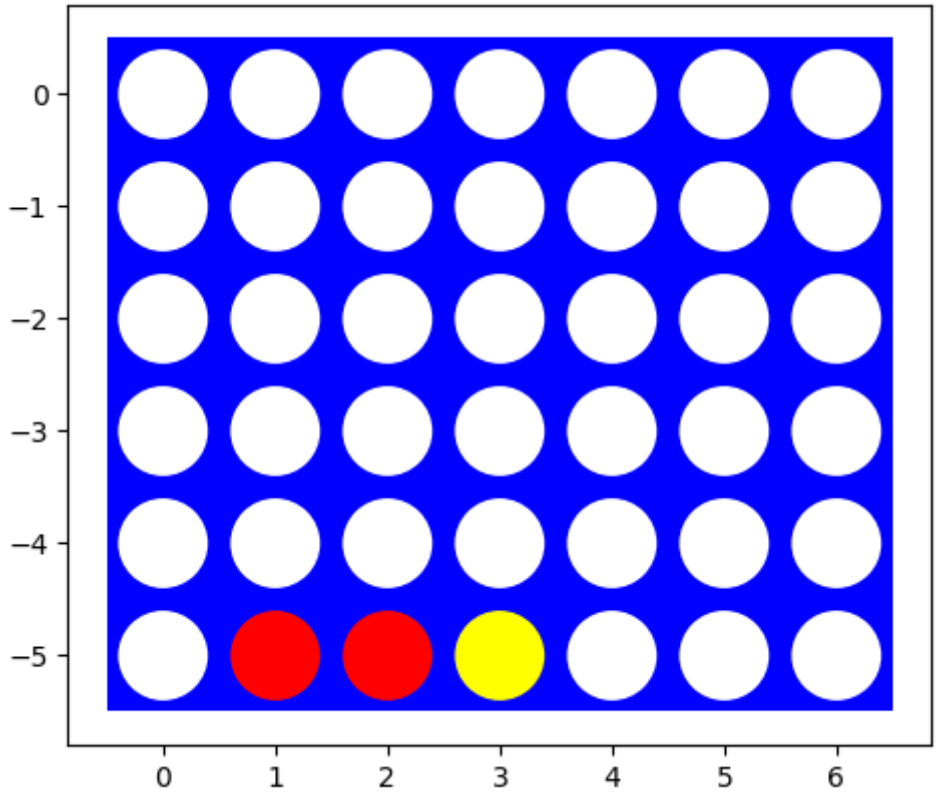


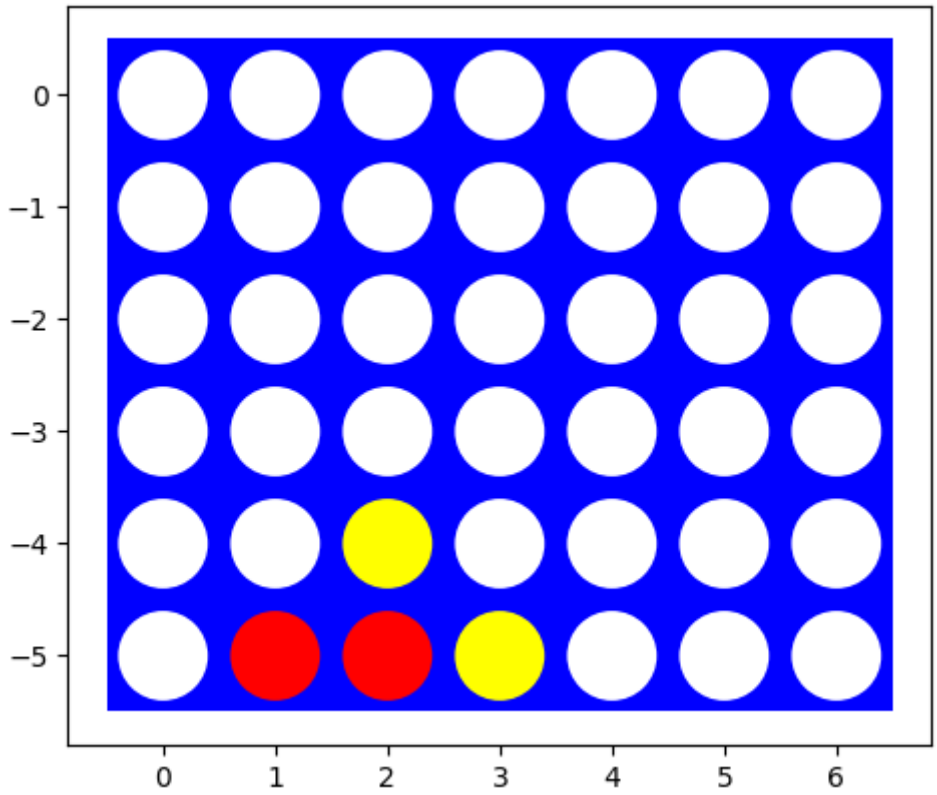

```
Enter the column:1
```



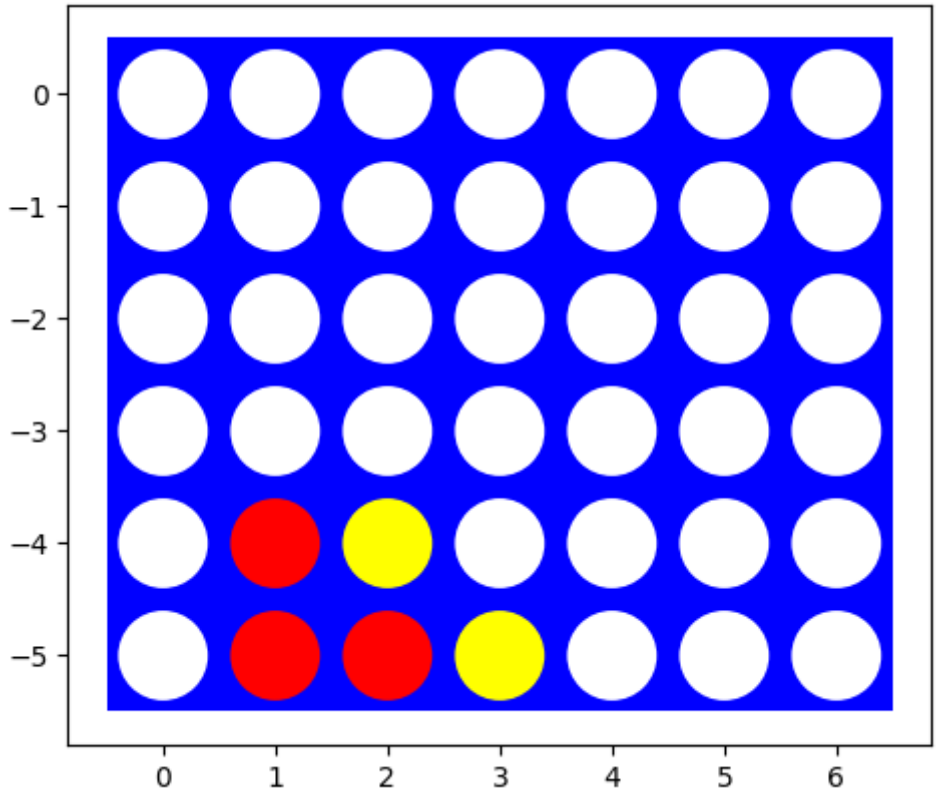


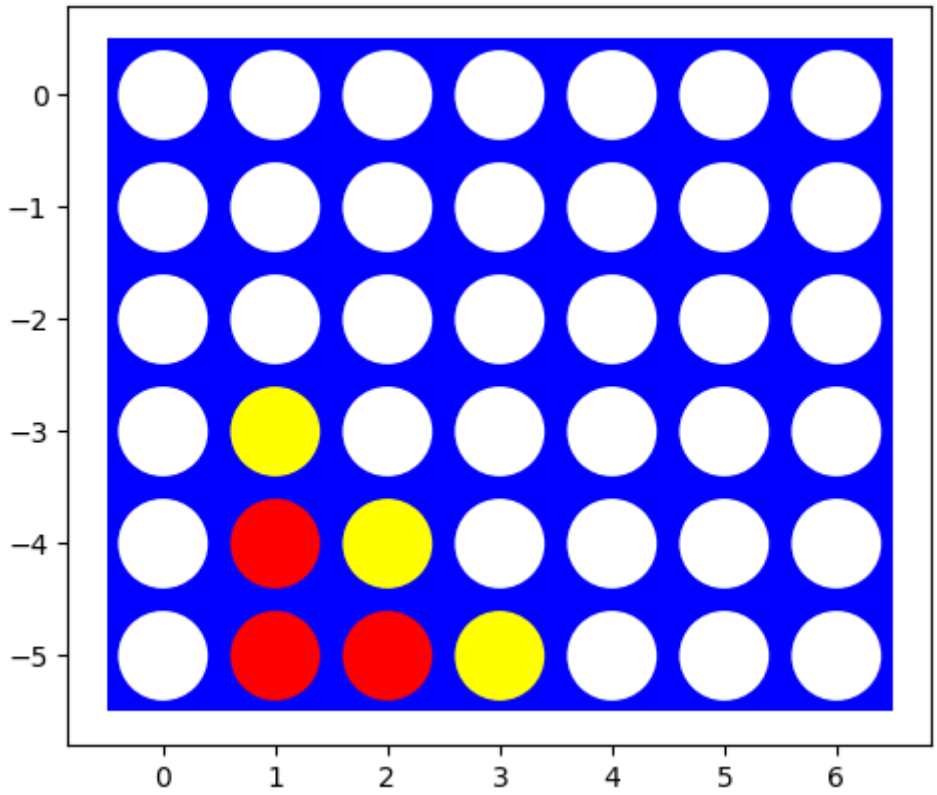

```
Enter the column:2
```



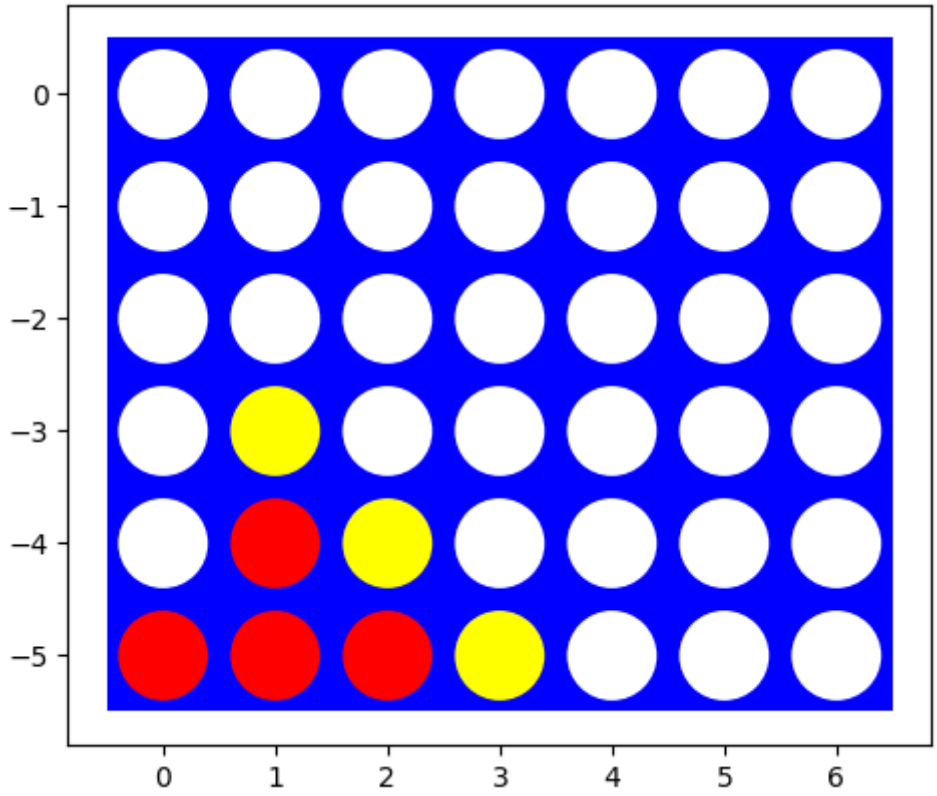


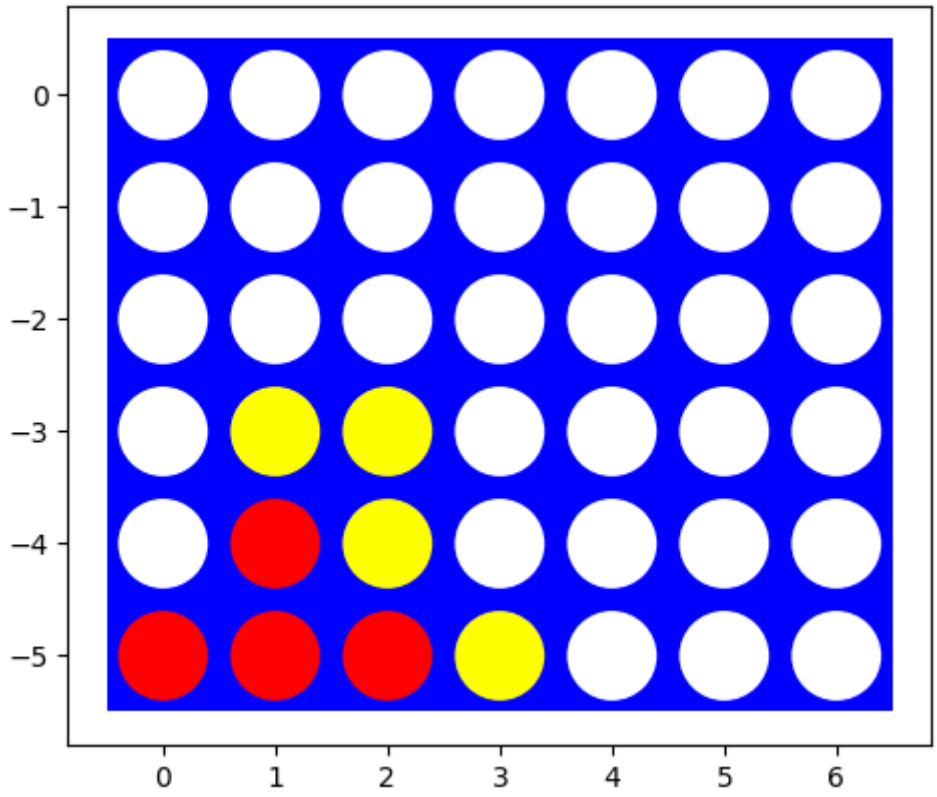

```
Enter the column:1
```



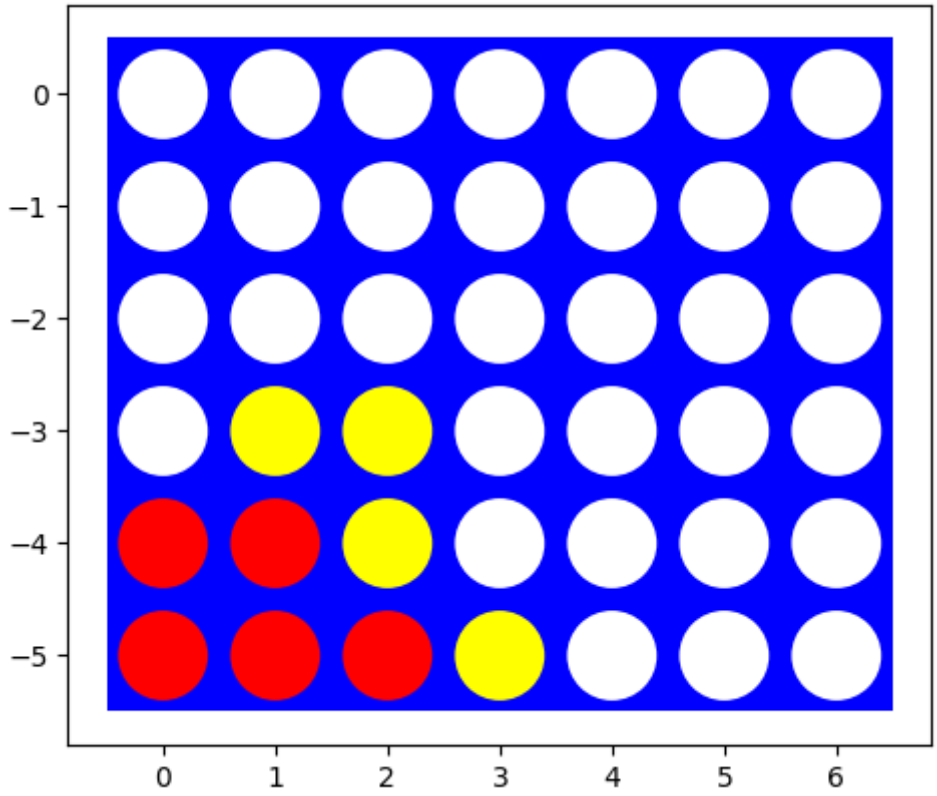


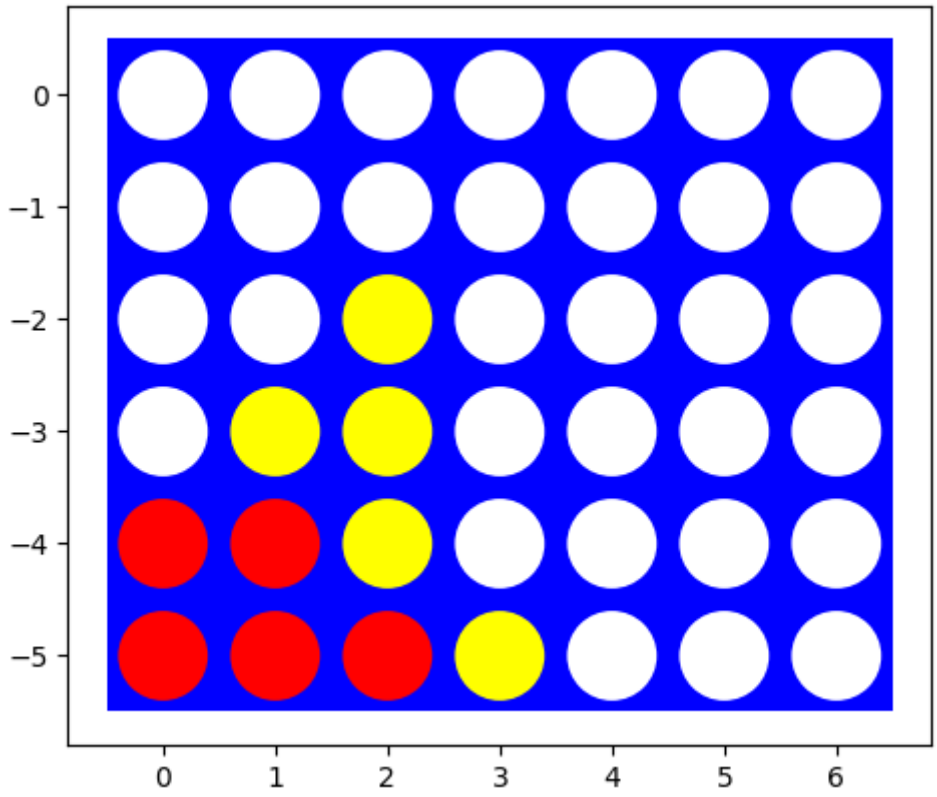

```
Enter the column:0
```



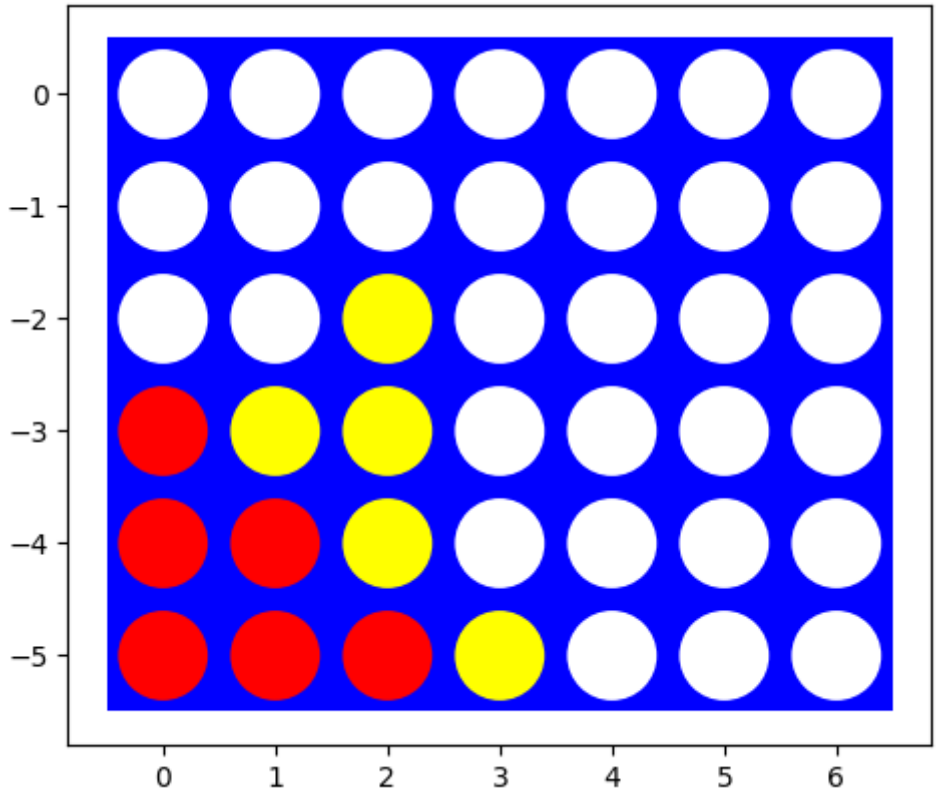



```
Enter the column:0
```

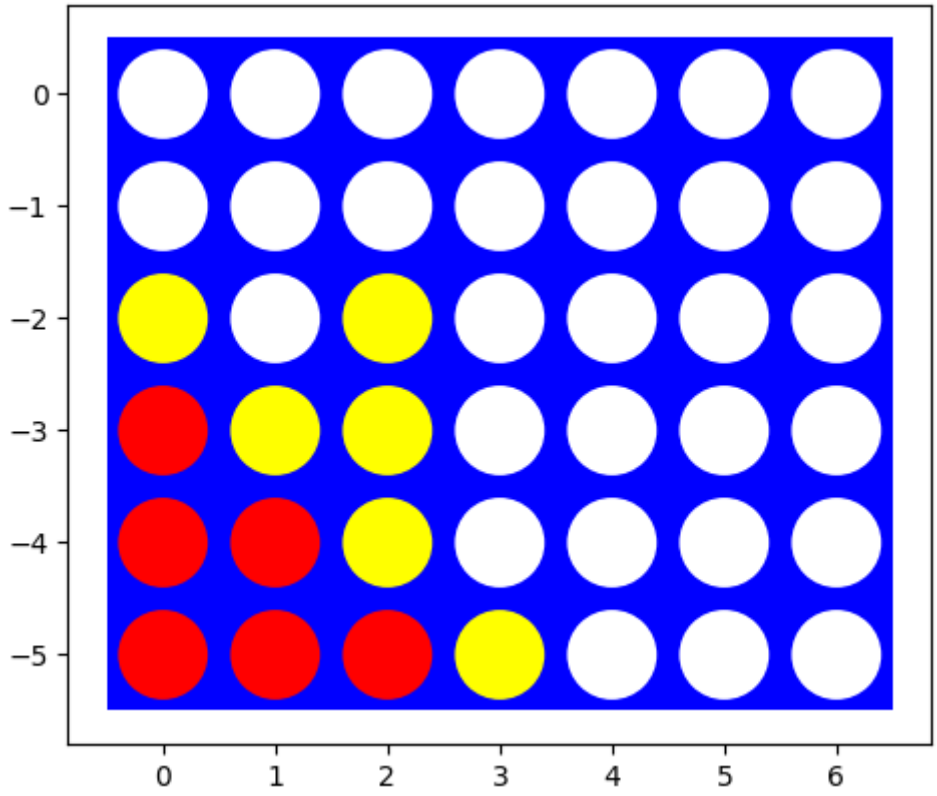





```
Enter the column:0
```



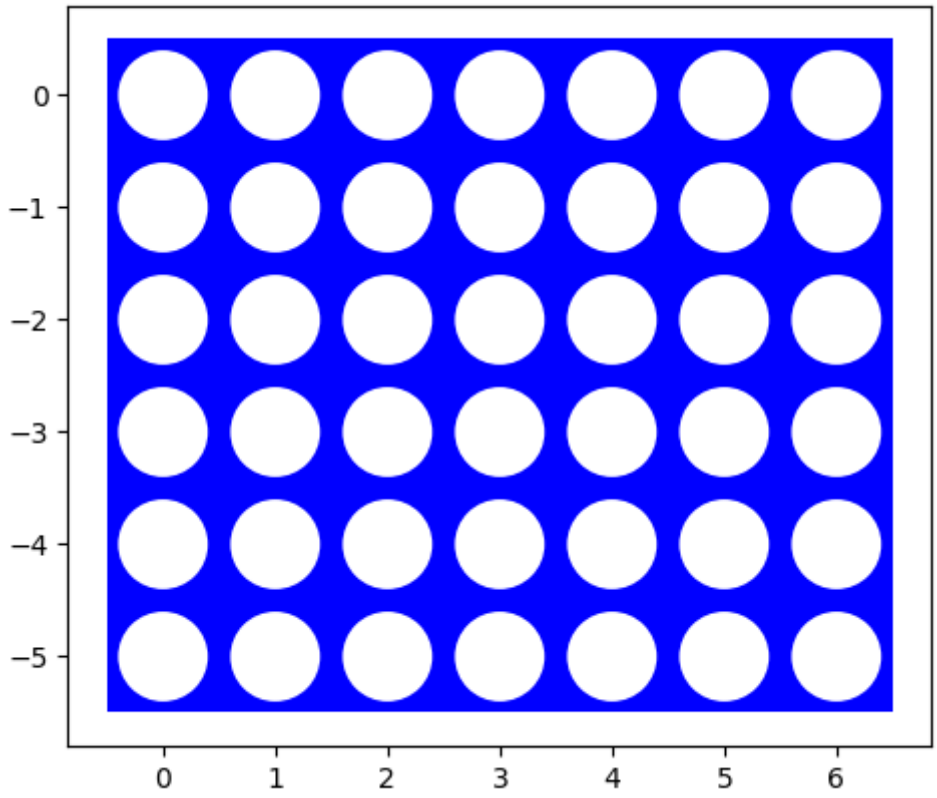
AI WIN!



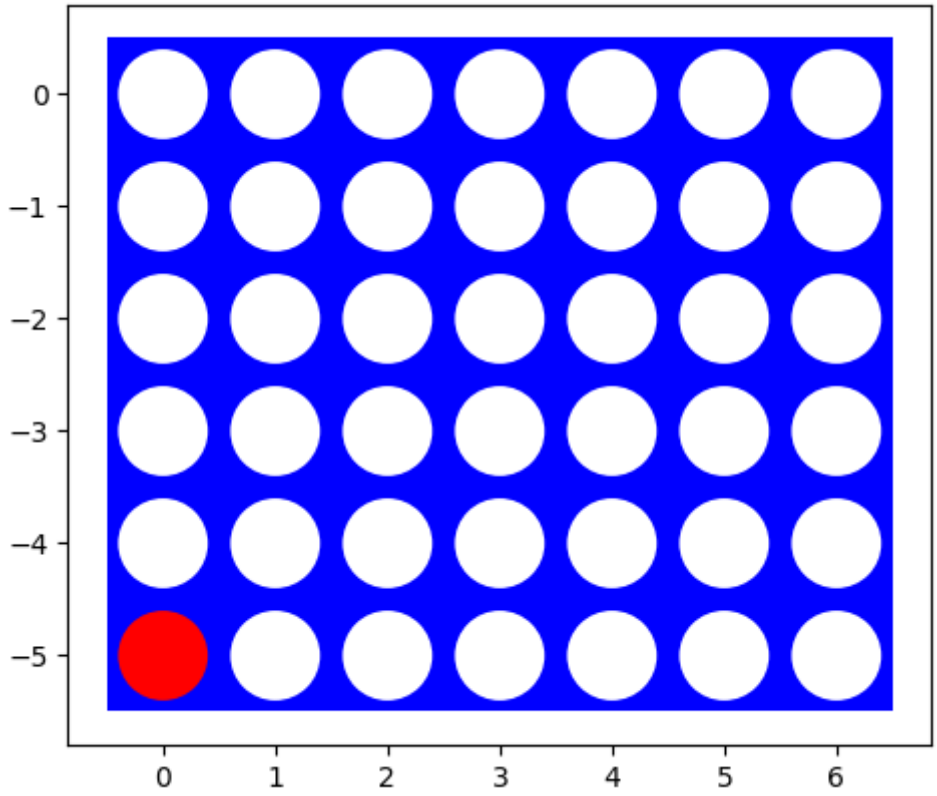
In [48]:

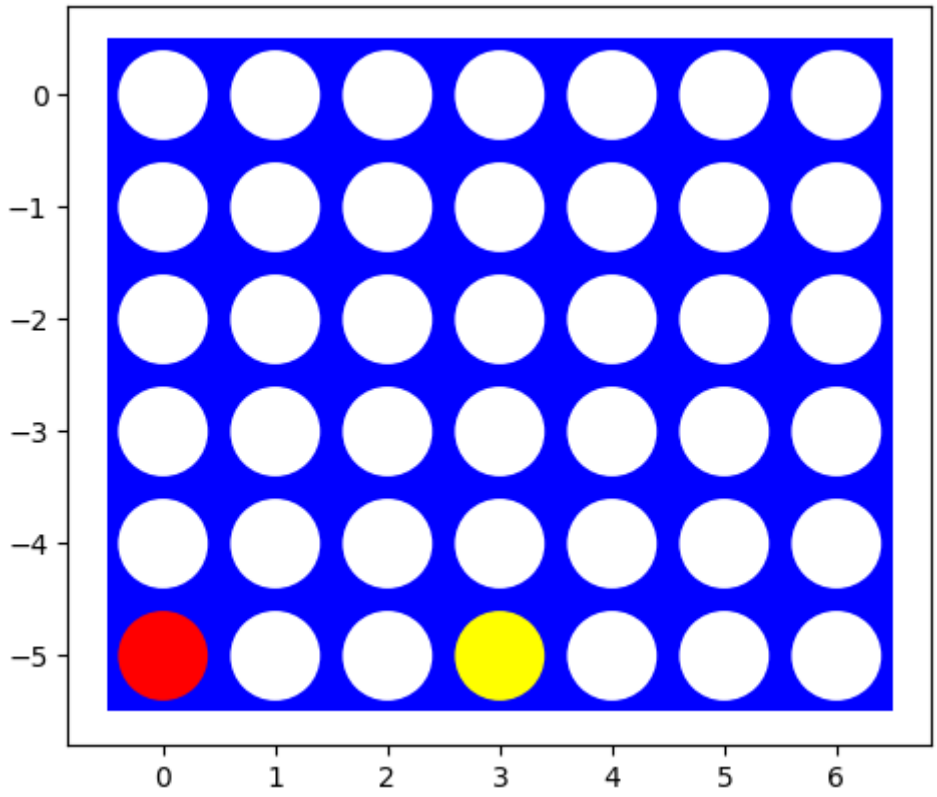
```
game_over_check = False
turn = 0  # Human turn is first
difficulty = 0


board = initialize_board(6,7)
visualize(board)

while not game_over_check:
    if turn == 0:
```

```
        col = int(input("Enter the column:"))
        if valid_actions_check(board, col):
            row = get_row(board, col)
            update_board(board, row, col, 1)
            #print_board(board)
            if check_winner(board,1):
                print("YOU WIN !")
                game_over_check = True
            turn += 1
            visualize(board)
    elif turn == 1:
        func_score, best_col = minmax(board, True, -math.inf, math.inf, 3)
        if valid_actions_check(board, best_col):
            row = get_row(board, best_col)
            update_board(board, row, best_col, 2)
            #print_board(board)
            if check_winner(board, 2):
                print("AI WIN!")
                game_over_check = True
            turn -= 1
            visualize(board)
```

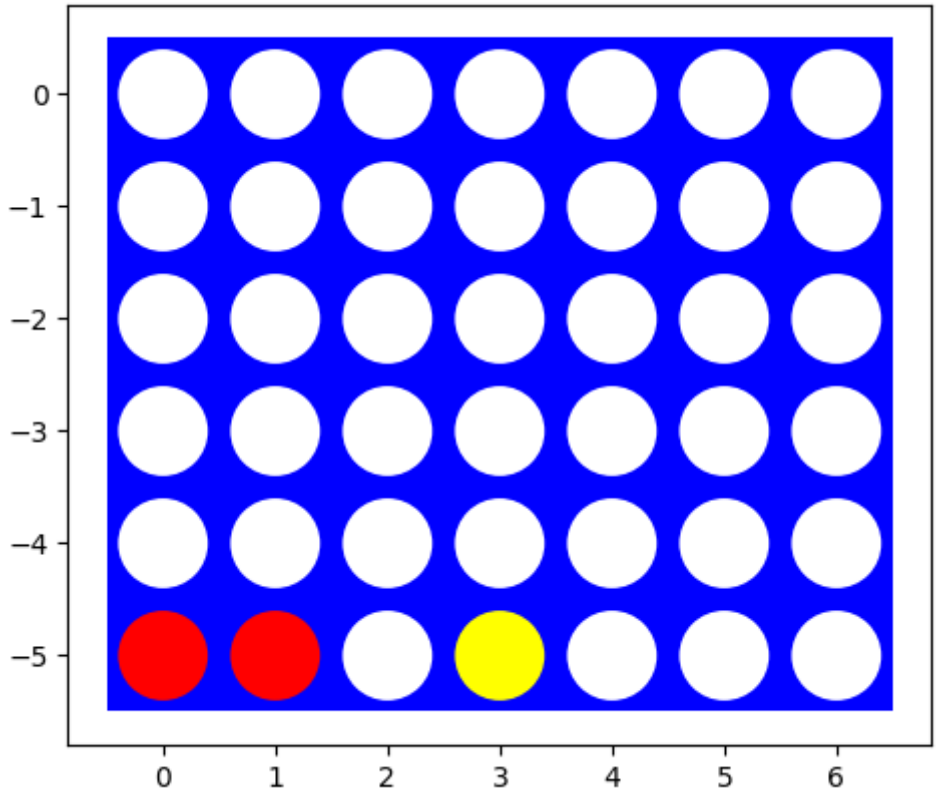


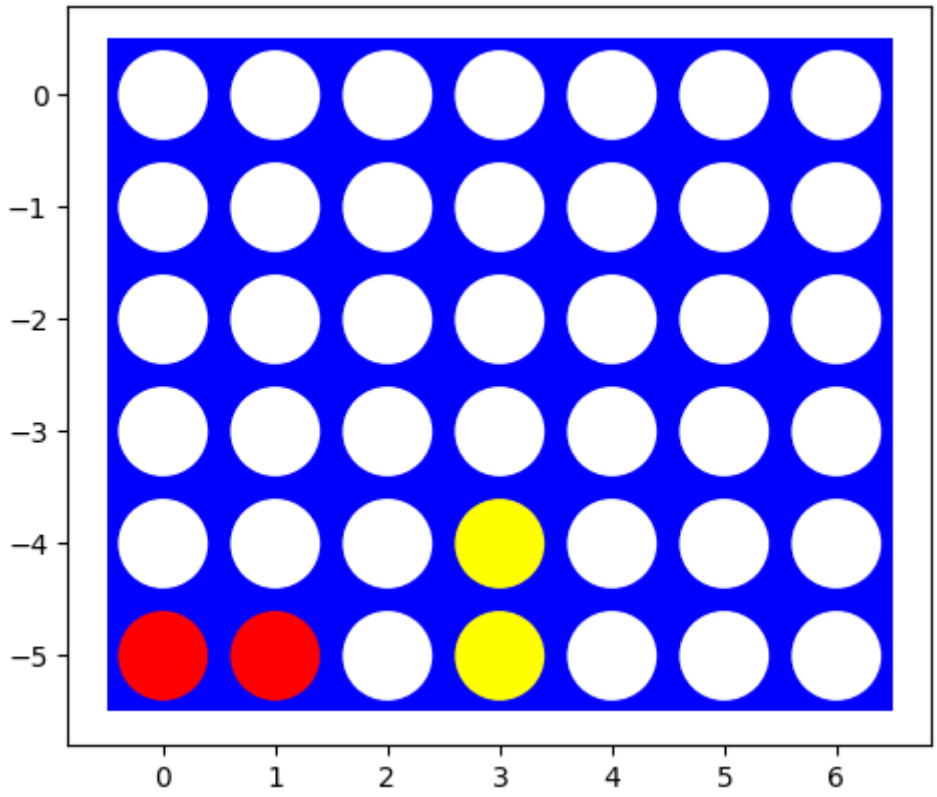

```
Enter the column:0
```



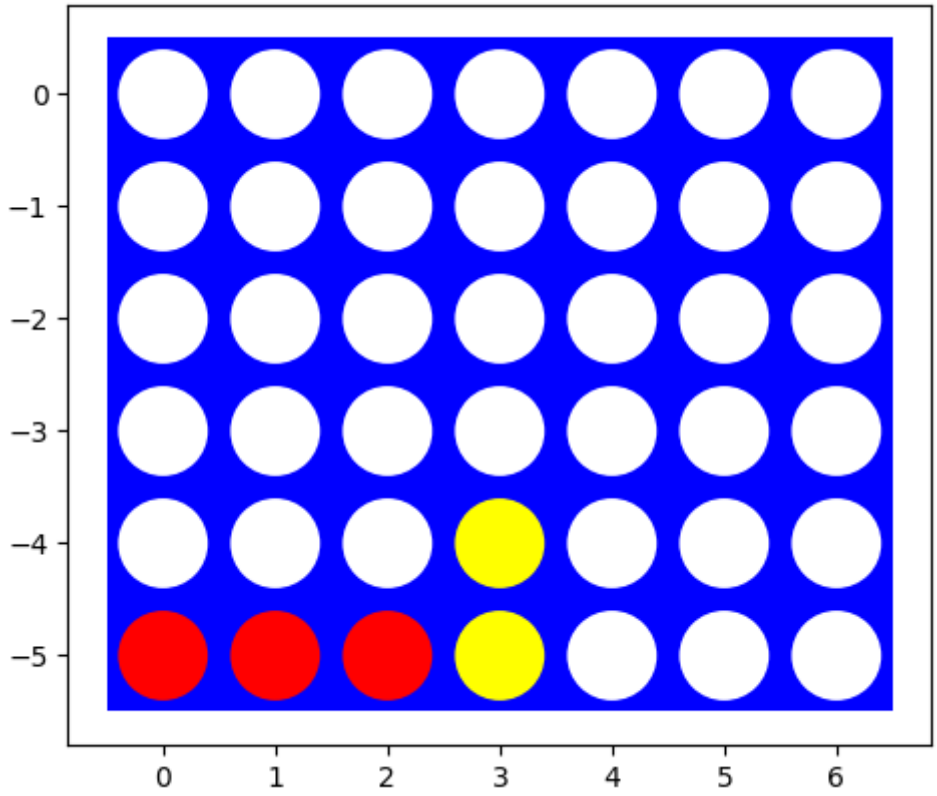


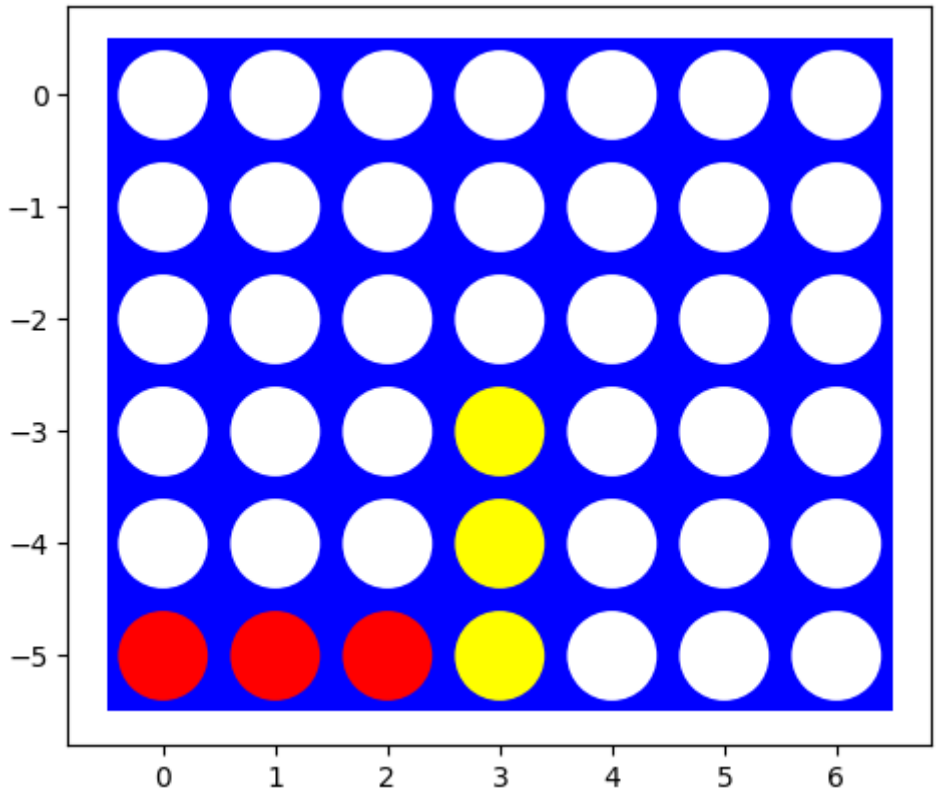

```
Enter the column:1
```



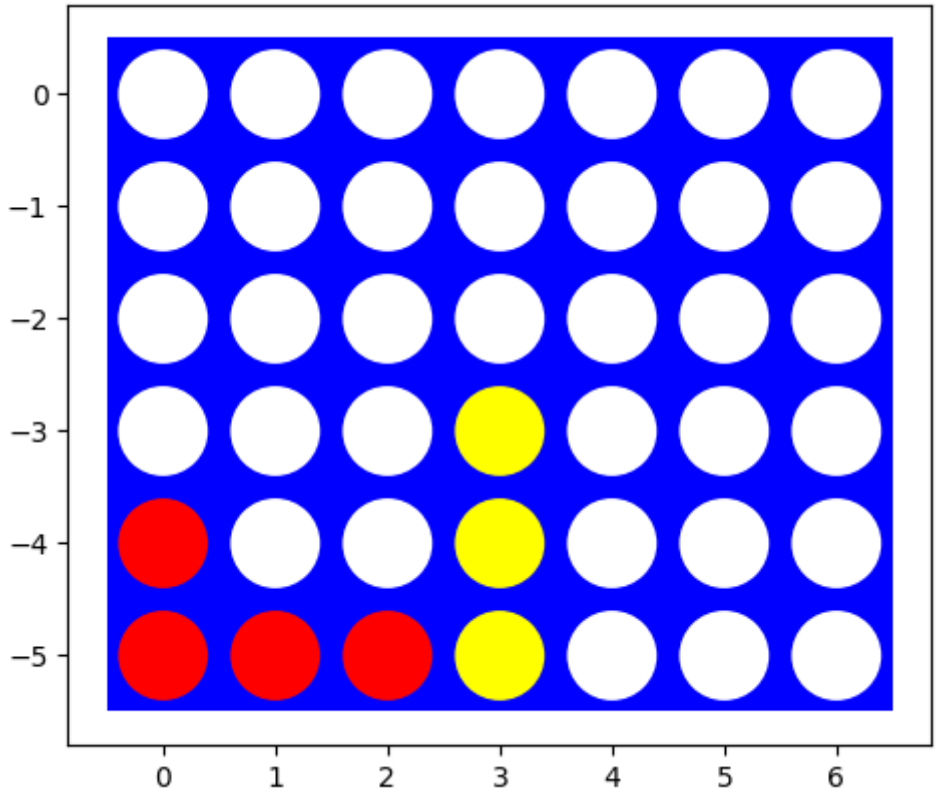



```
Enter the column:2
```

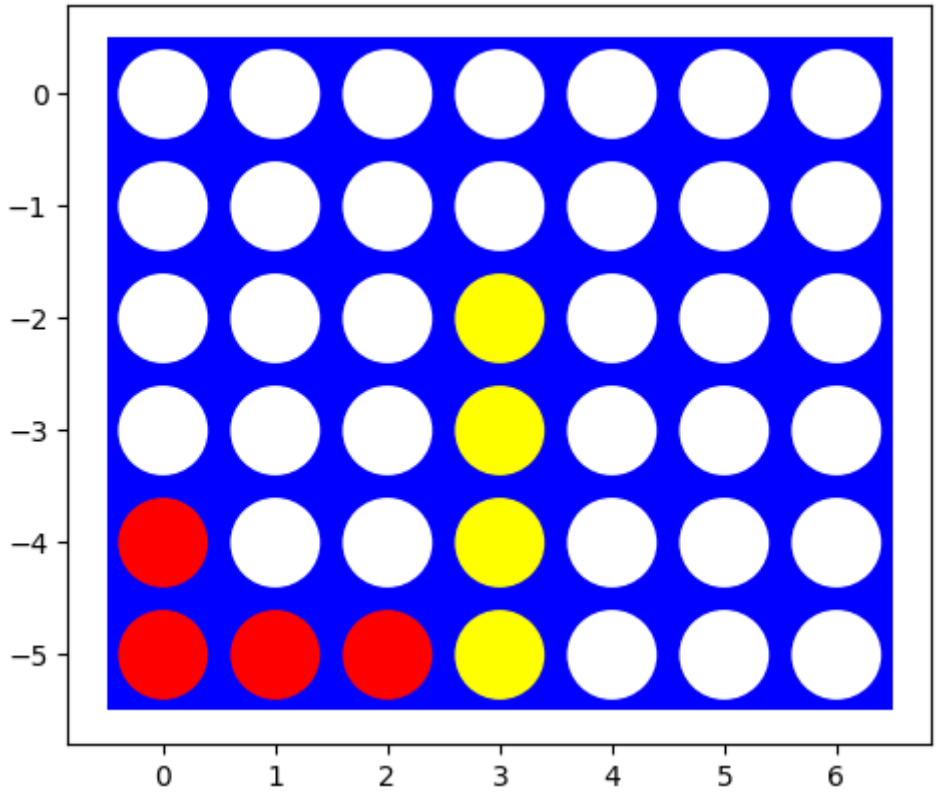





```
Enter the column:0
```



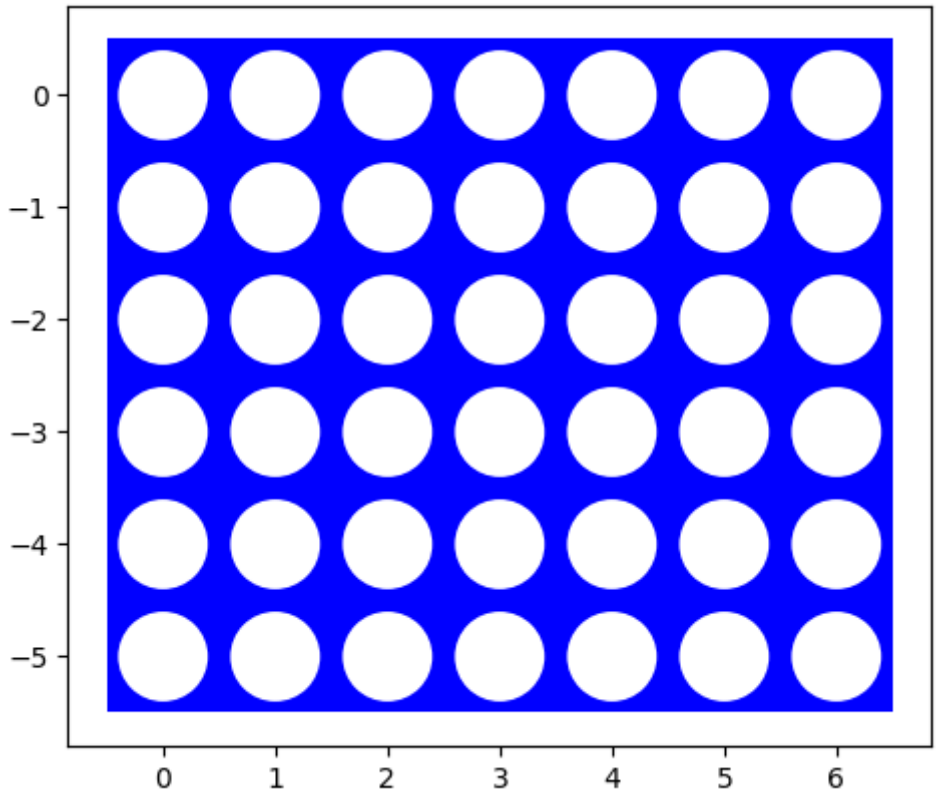
AI WIN!



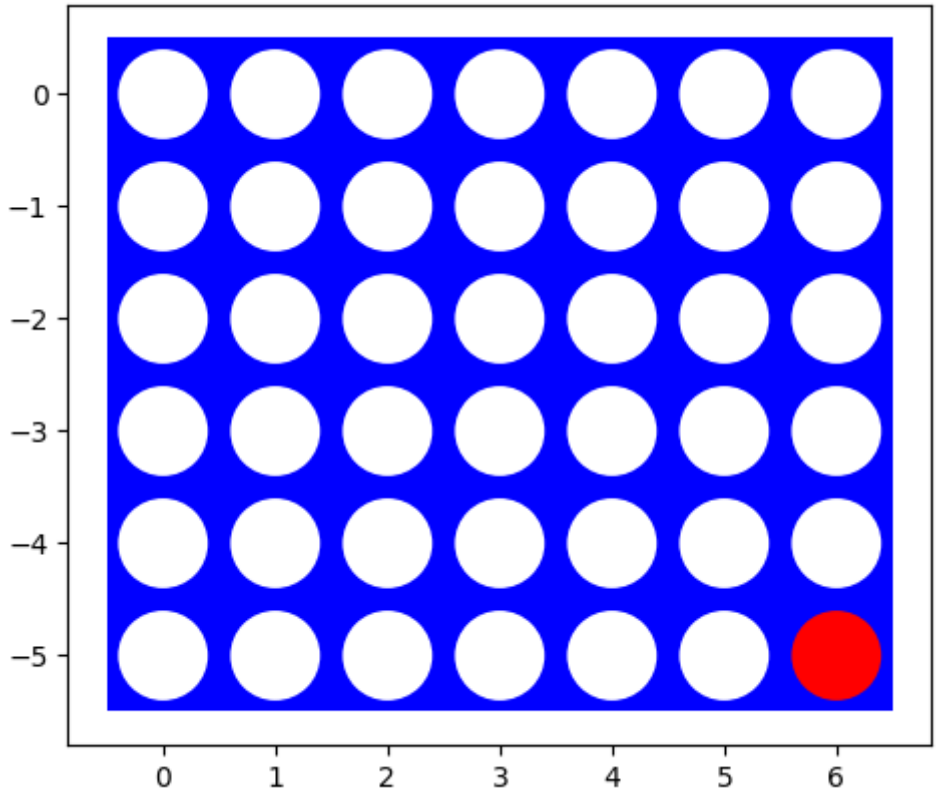
In [44]:

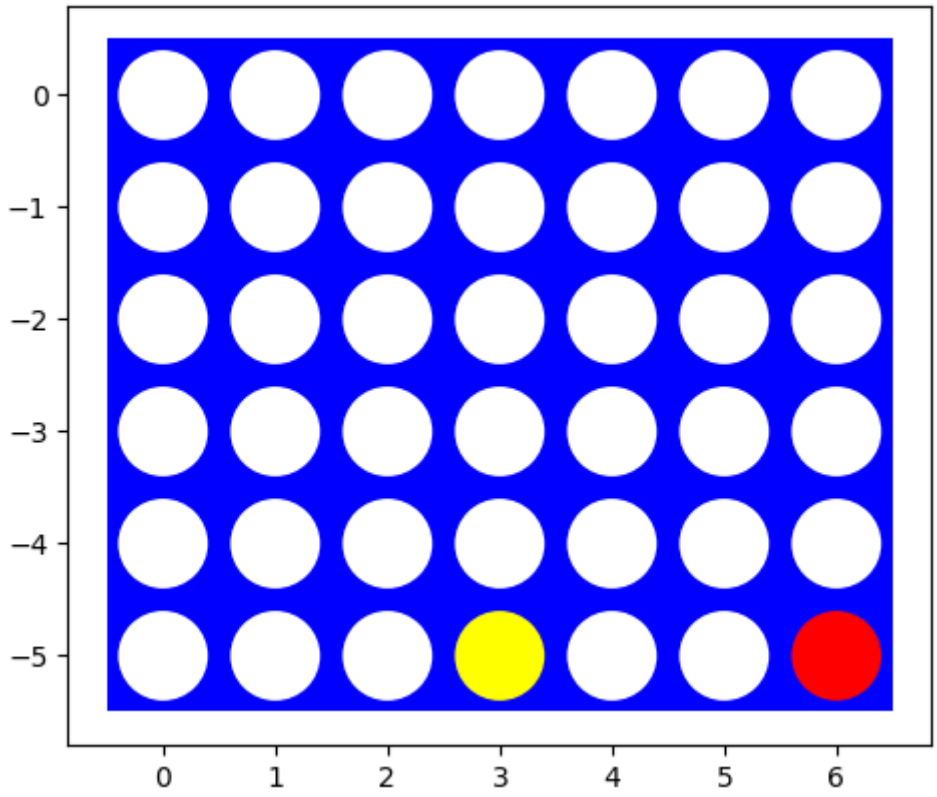
```
game_over_check = False
turn = 0  # Human turn is first
difficulty = 0


board = initialize_board(6,7)
visualize(board)

while not game_over_check:
    if turn == 0:
```

```
        col = int(input("Enter the column:"))
        if valid_actions_check(board, col):
            row = get_row(board, col)
            update_board(board, row, col, 1)
            #print_board(board)
            if check_winner(board,1):
                print("YOU WIN !")
                game_over_check = True
            turn += 1
            visualize(board)
    elif turn == 1:
        func_score, best_col = minmax(board, True, -math.inf, math.inf, 3)
        if valid_actions_check(board, best_col):
            row = get_row(board, best_col)
            update_board(board, row, best_col, 2)
            #print_board(board)
            if check_winner(board, 2):
                print("AI WIN!")
                game_over_check = True
            turn -= 1
            visualize(board)
```

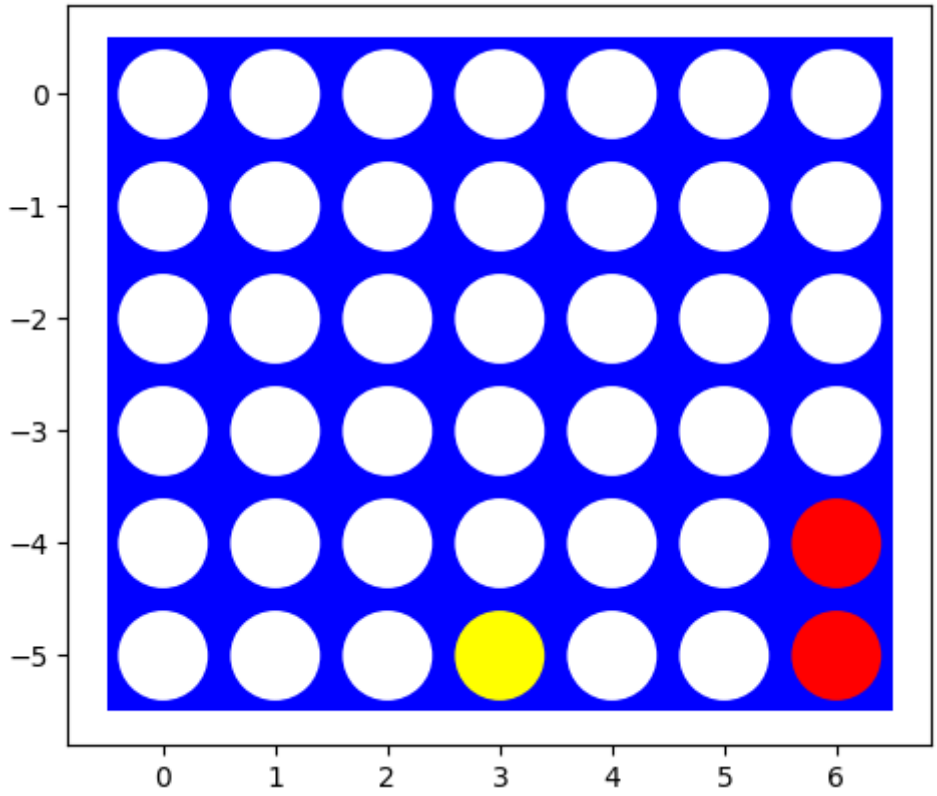


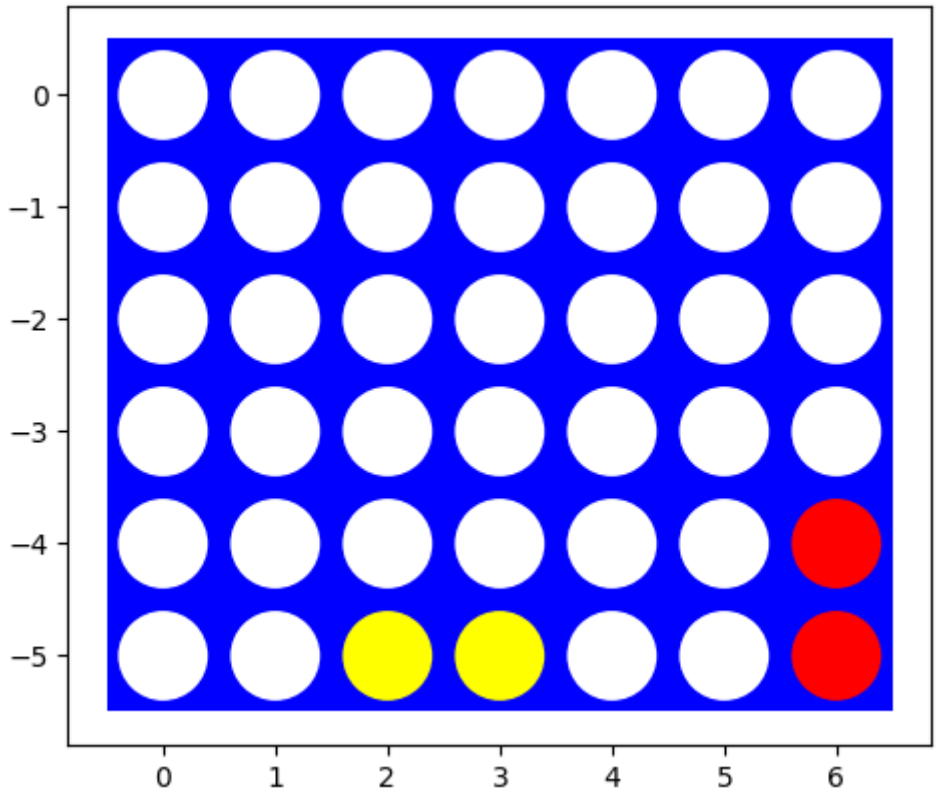

```
Enter the column:6
```



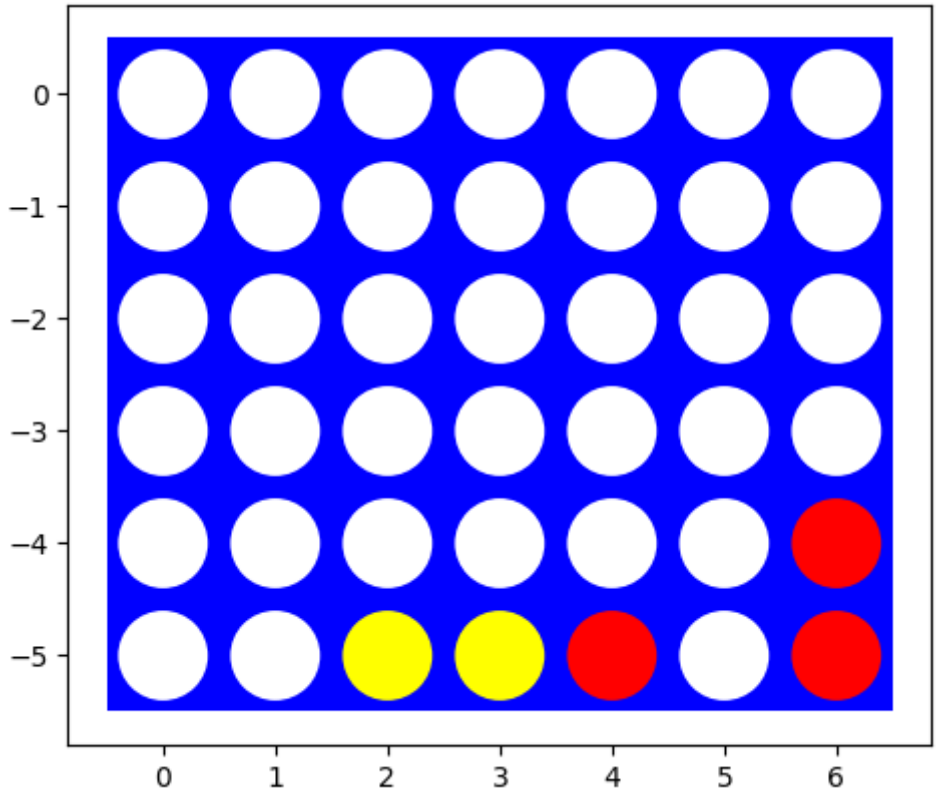


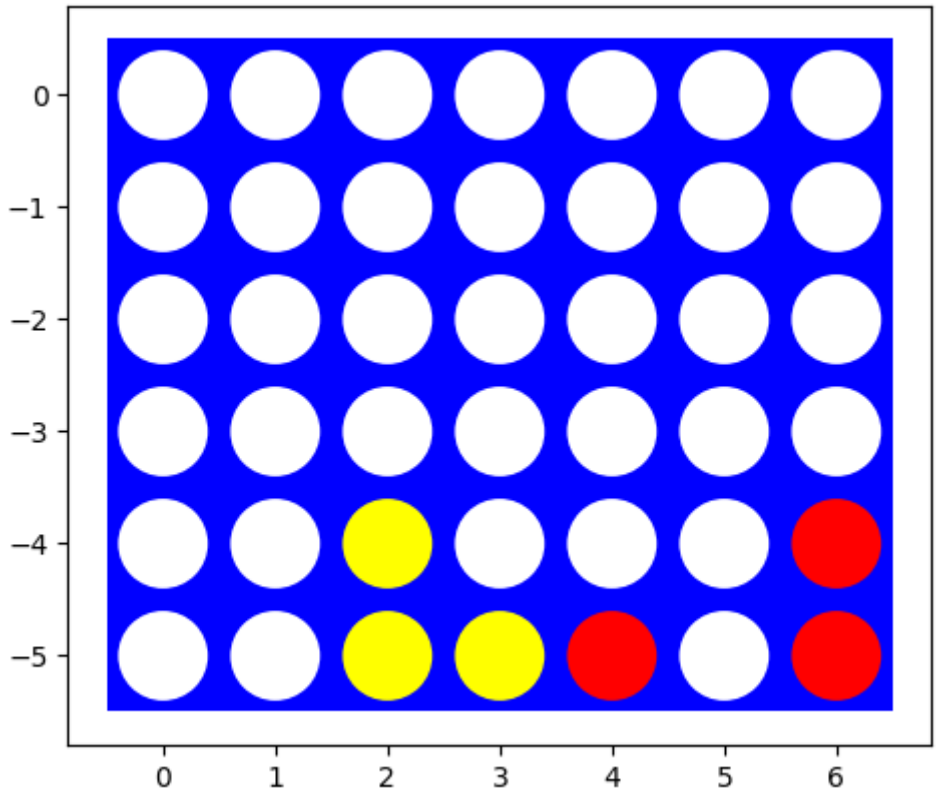

```
Enter the column:6
```



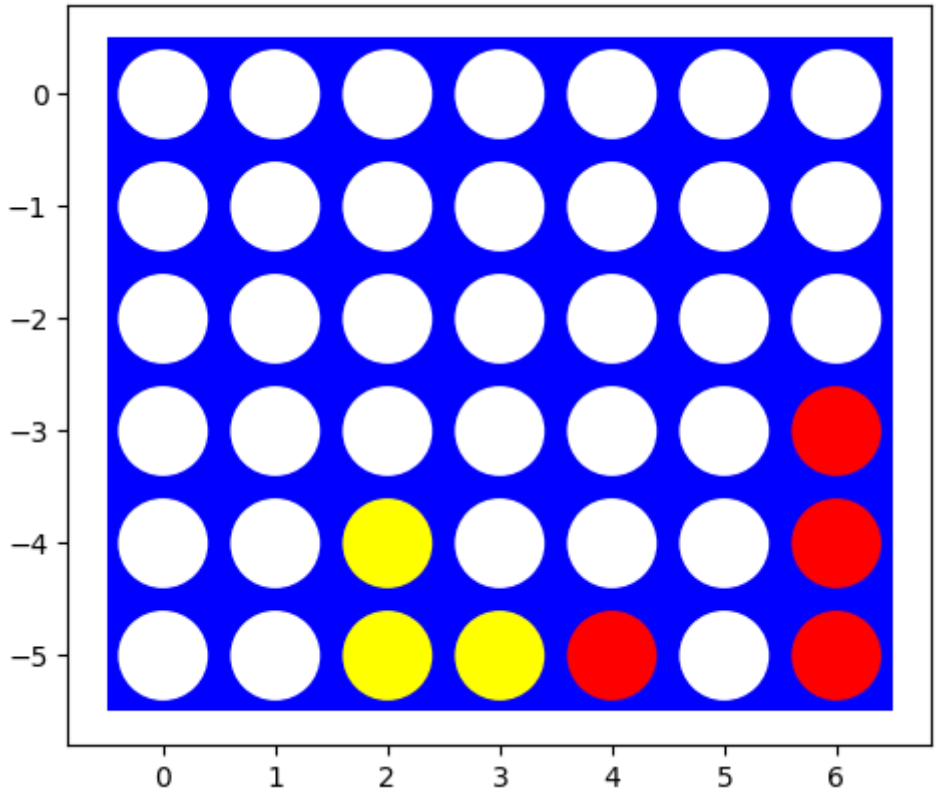


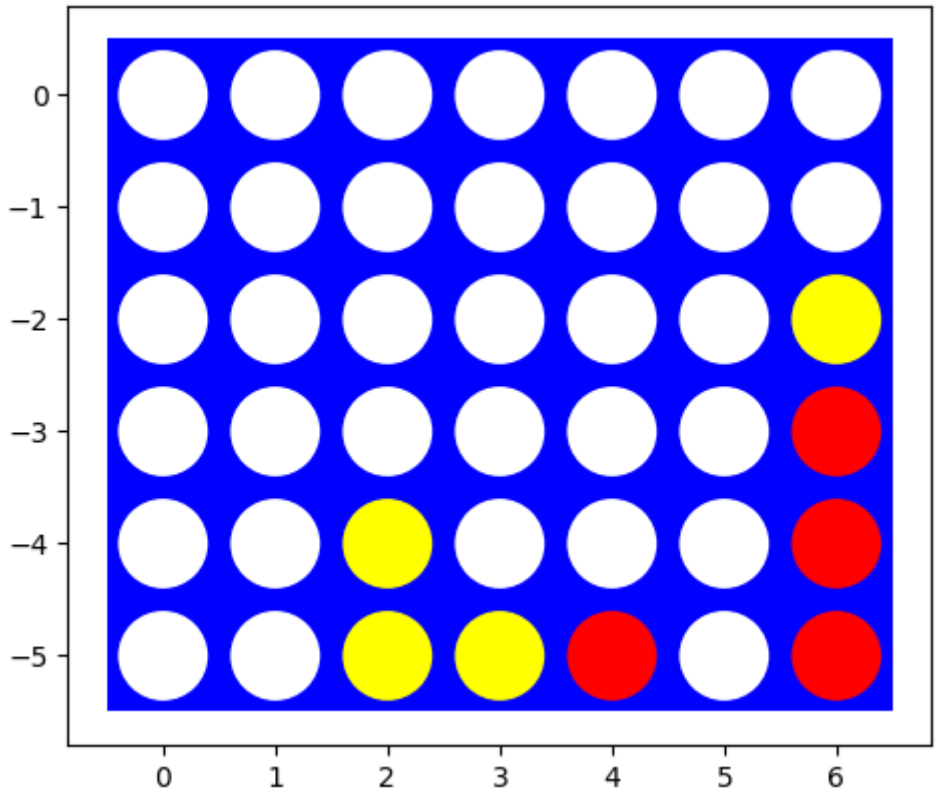

```
Enter the column:4
```



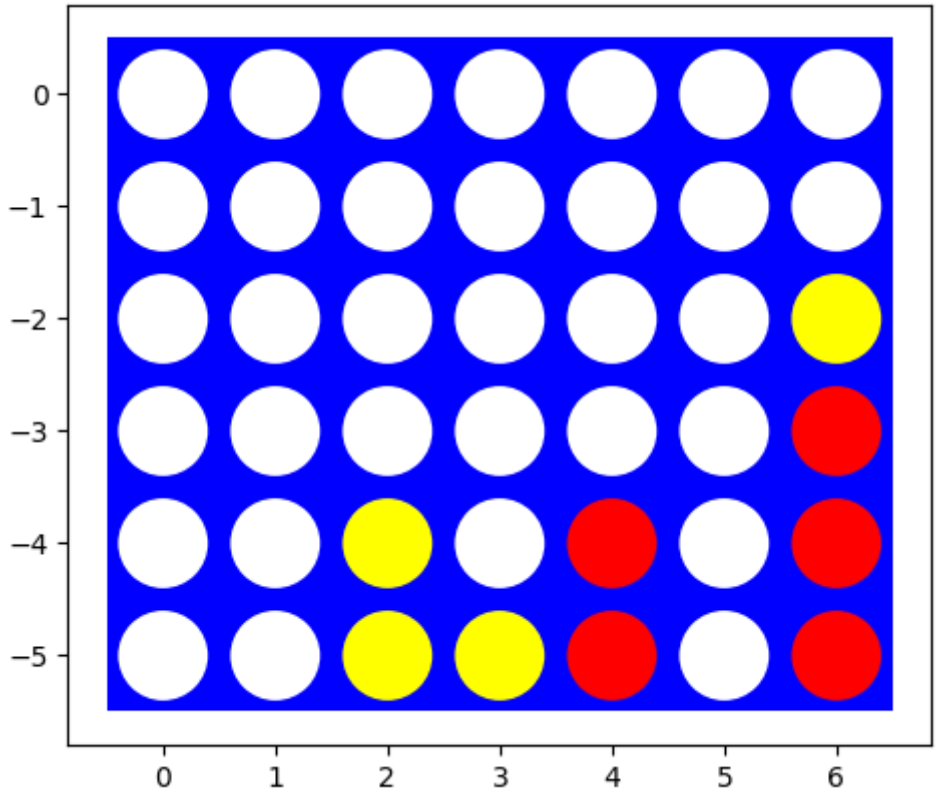


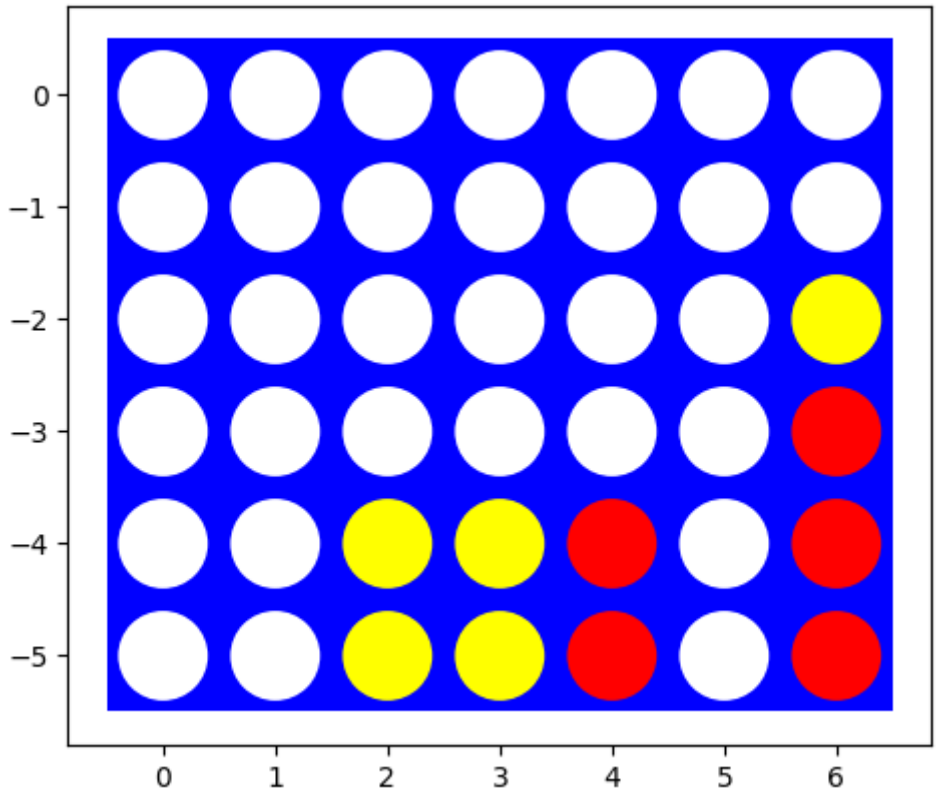

```
Enter the column:6
```



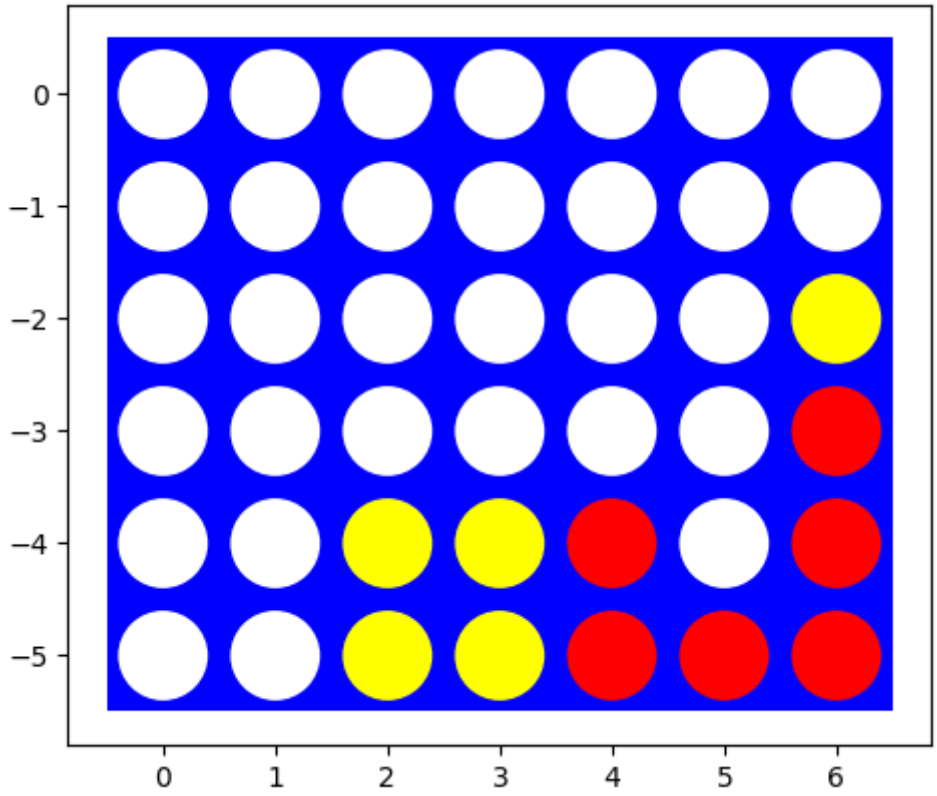


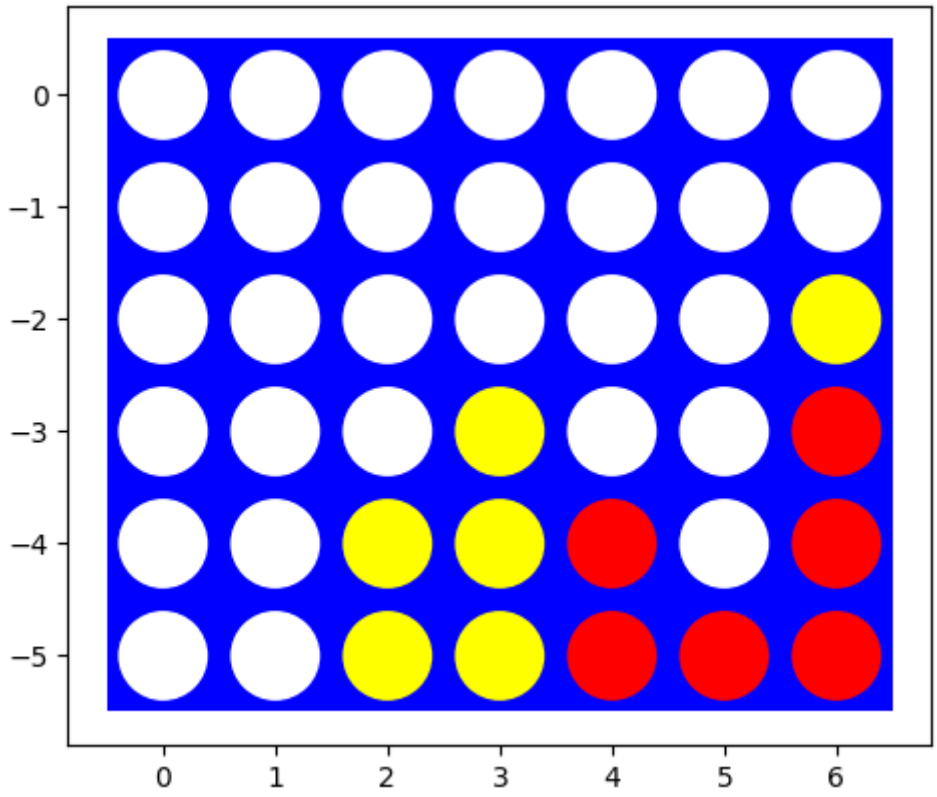

```
Enter the column:4
```



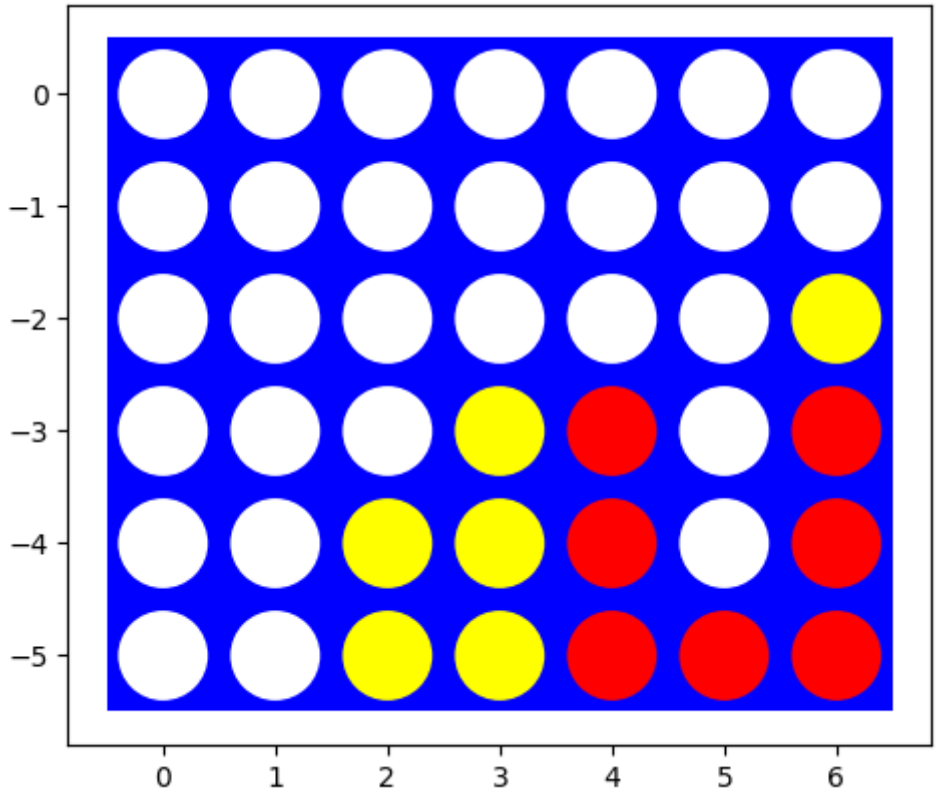


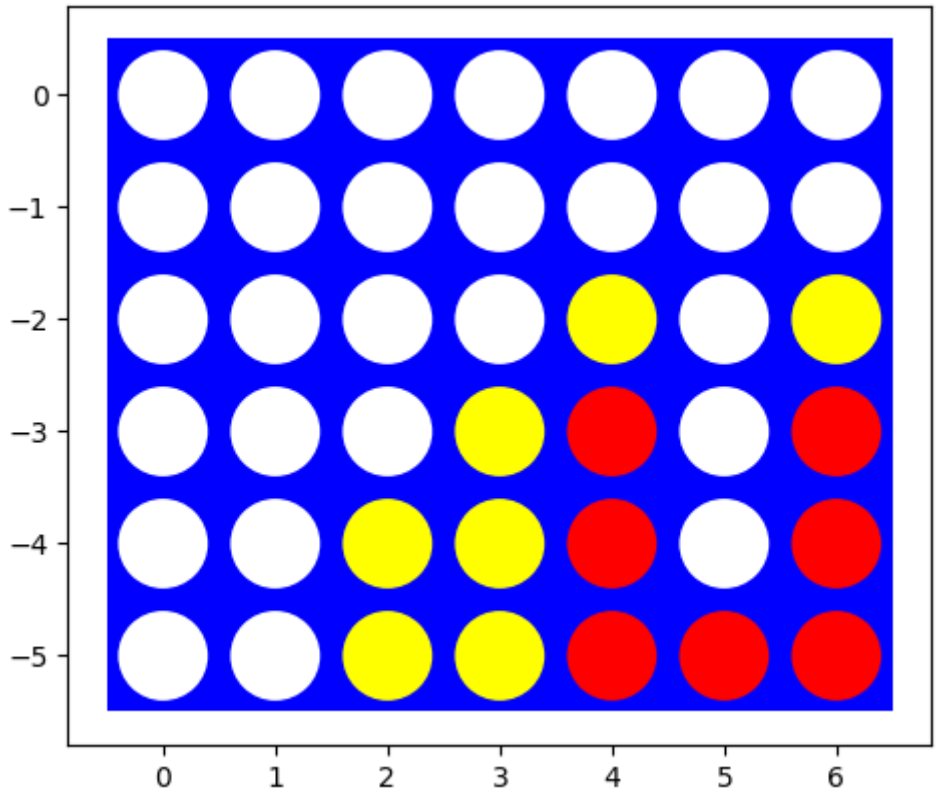

```
Enter the column:5
```



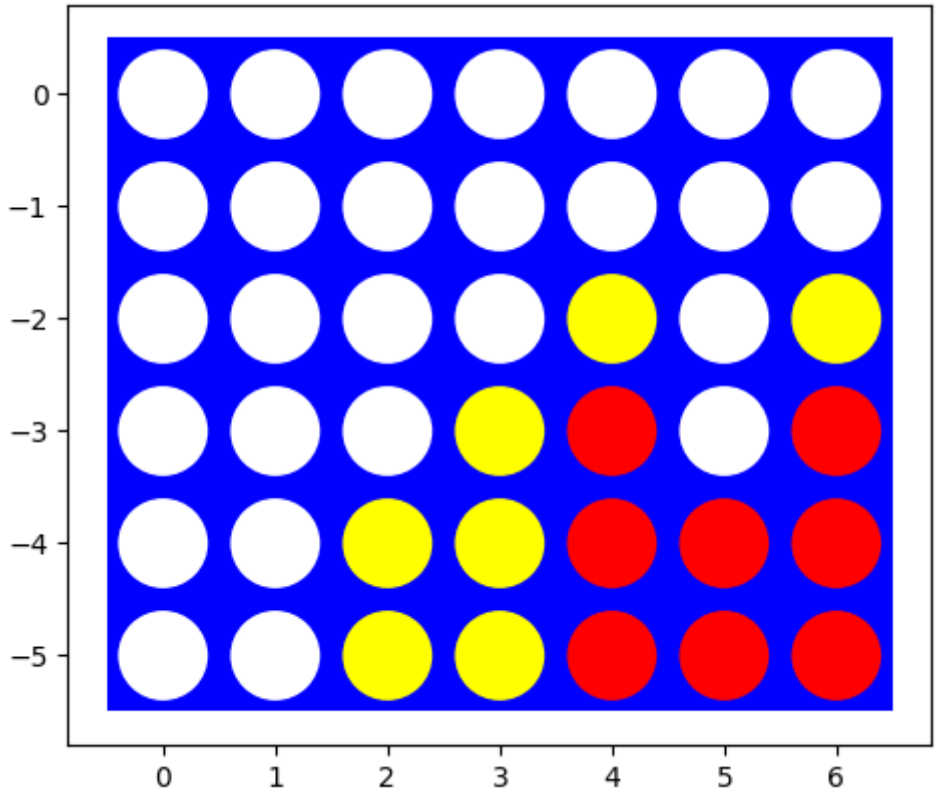



```
Enter the column:4
```

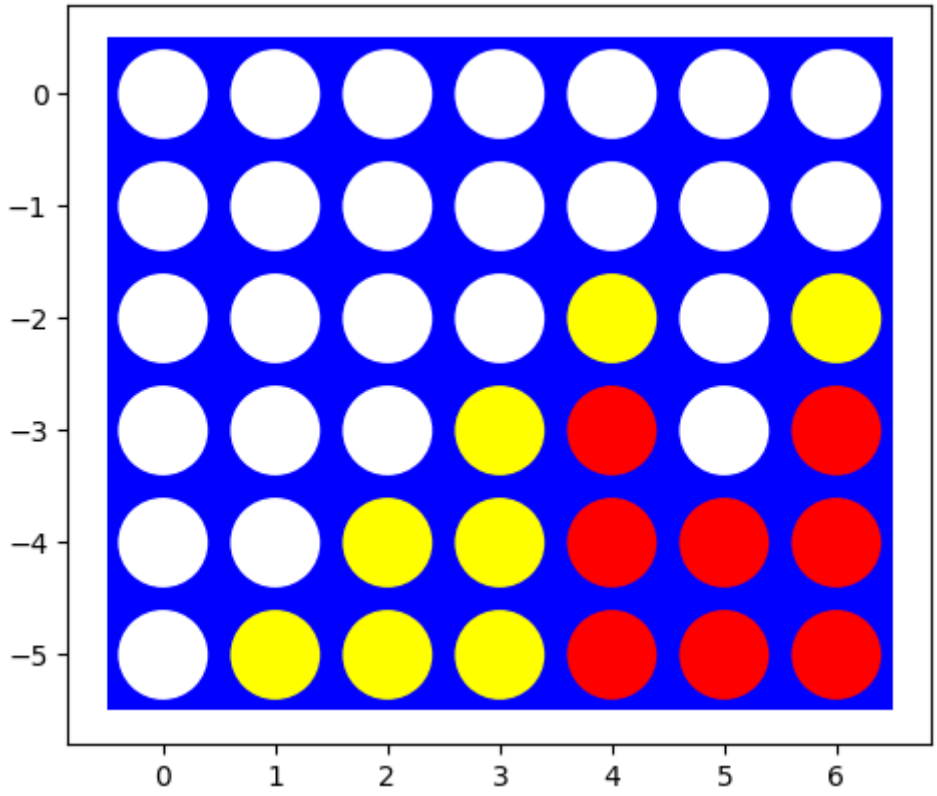





```
Enter the column:5
```



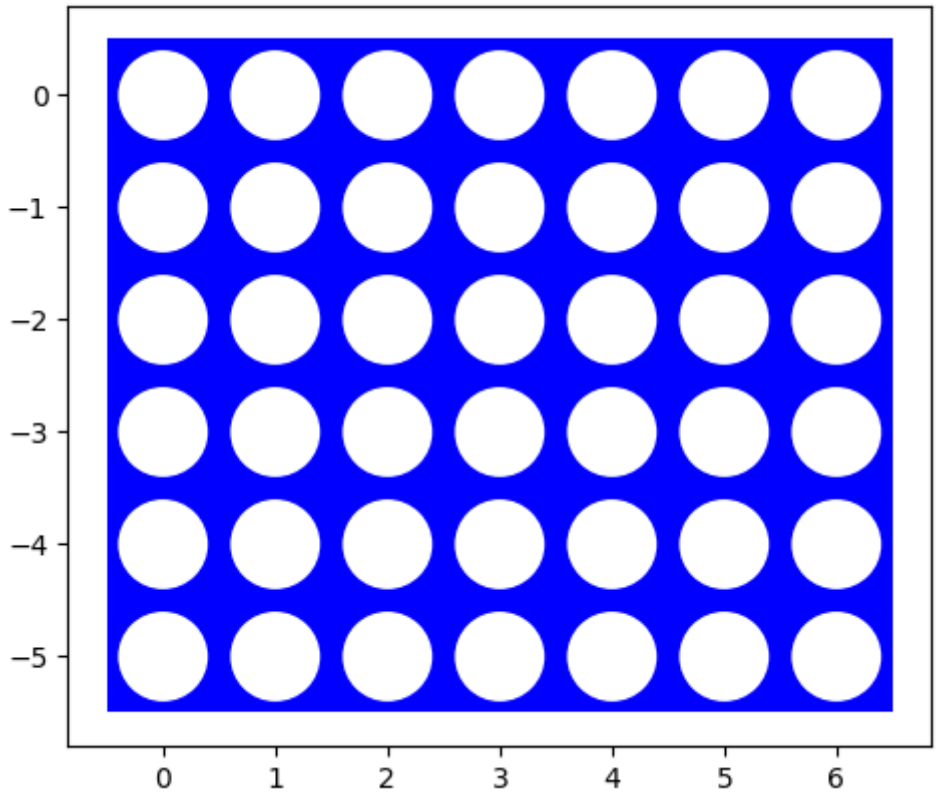
AI WIN!



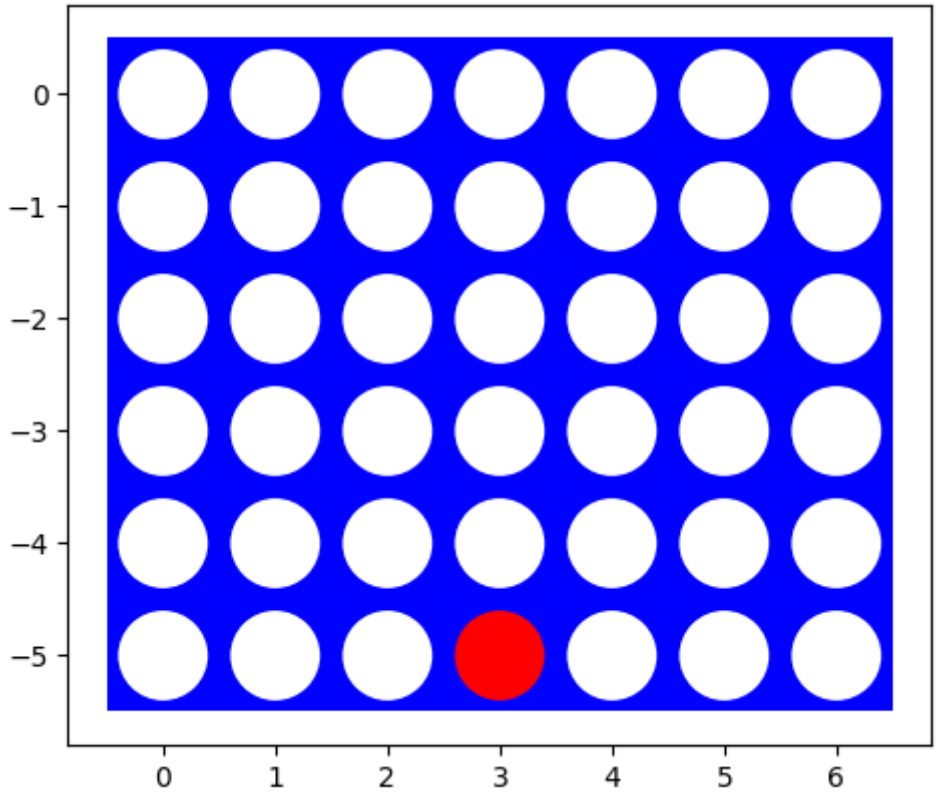
In [45]:

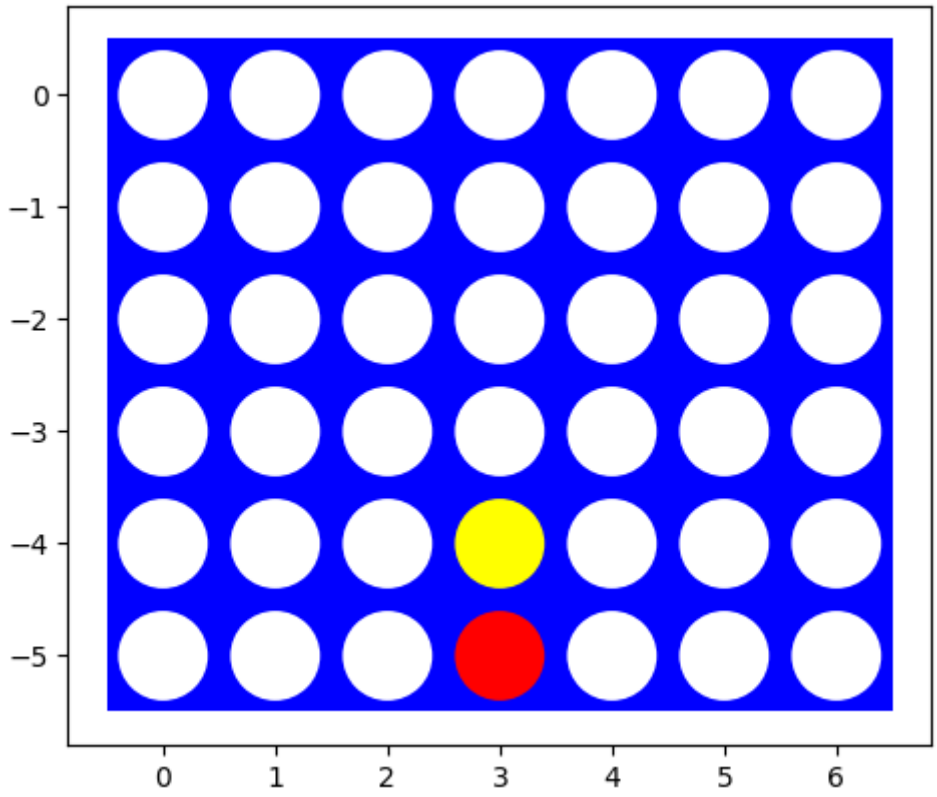
```
game_over_check = False
turn = 0  # Human turn is first
difficulty = 0


board = initialize_board(6,7)
visualize(board)

while not game_over_check:
    if turn == 0:
```

```
        col = int(input("Enter the column:"))
        if valid_actions_check(board, col):
            row = get_row(board, col)
            update_board(board, row, col, 1)
            #print_board(board)
            if check_winner(board,1):
                print("YOU WIN !")
                game_over_check = True
            turn += 1
            visualize(board)
    elif turn == 1:
        func_score, best_col = minmax(board, True, -math.inf, math.inf, 3)
        if valid_actions_check(board, best_col):
            row = get_row(board, best_col)
            update_board(board, row, best_col, 2)
            #print_board(board)
            if check_winner(board, 2):
                print("AI WIN!")
                game_over_check = True
            turn -= 1
            visualize(board)
```

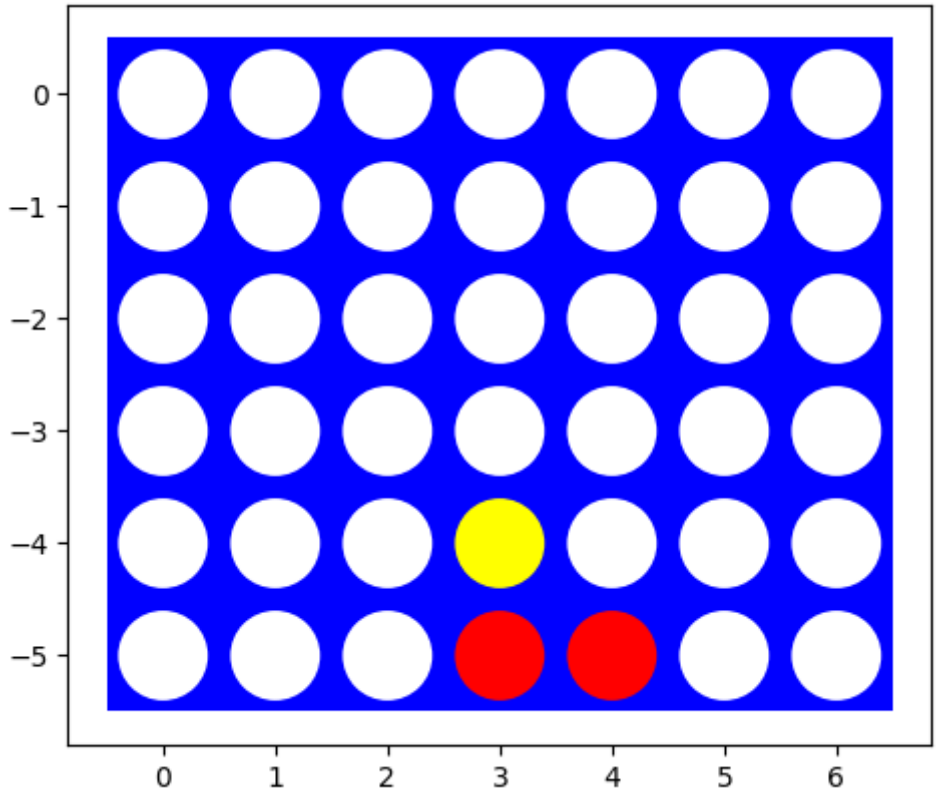


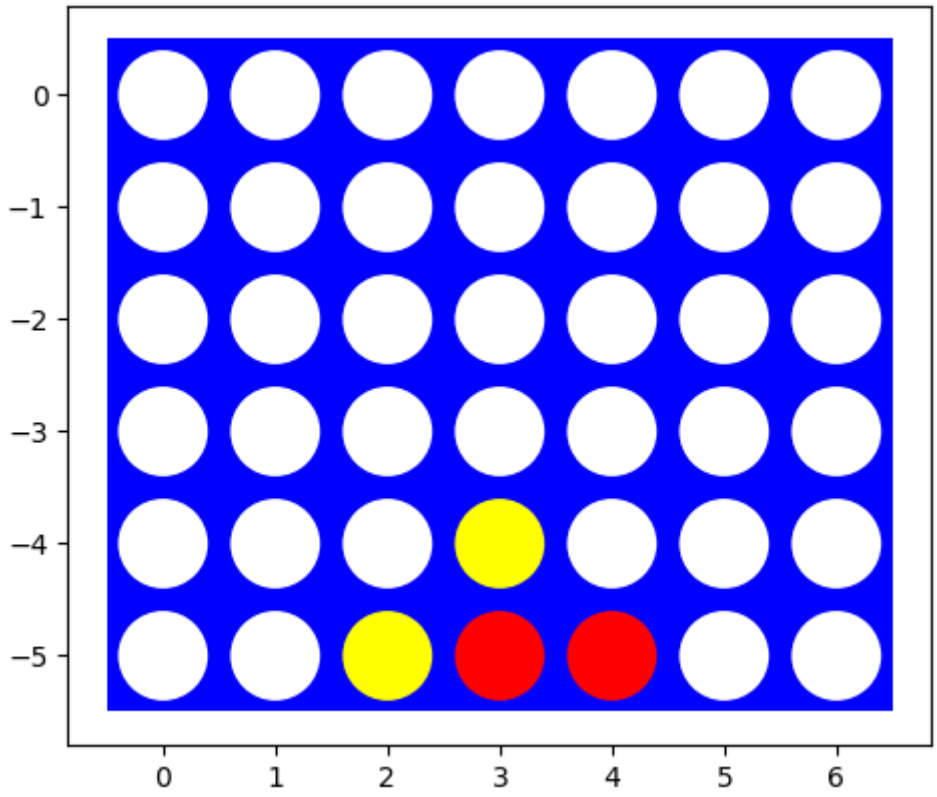

```
Enter the column:3
```



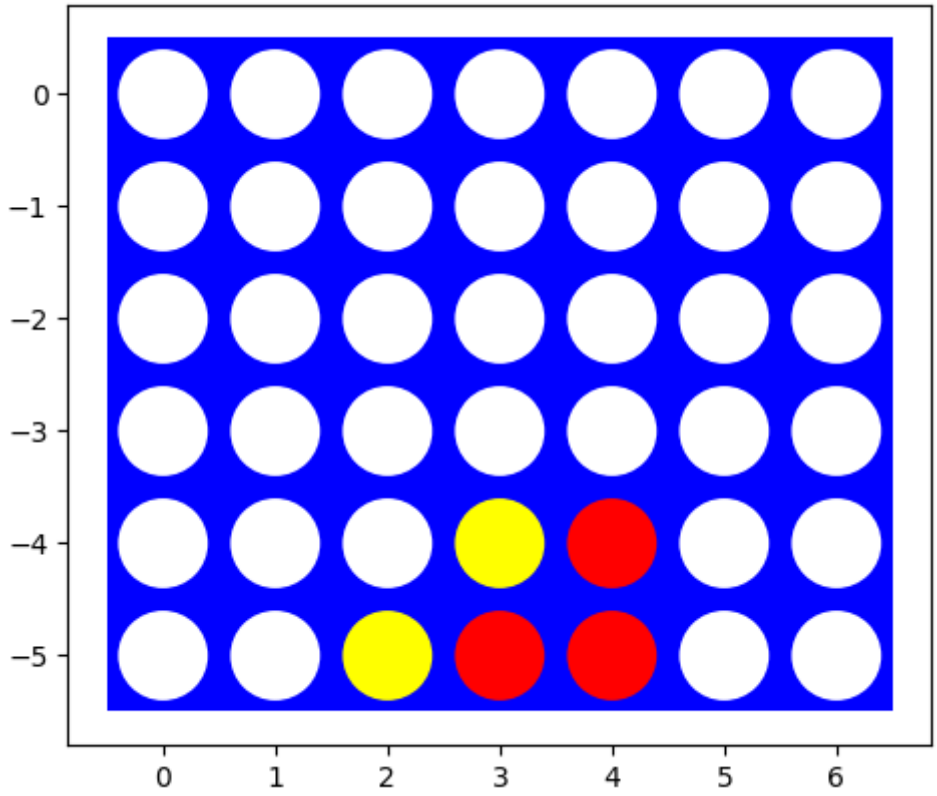


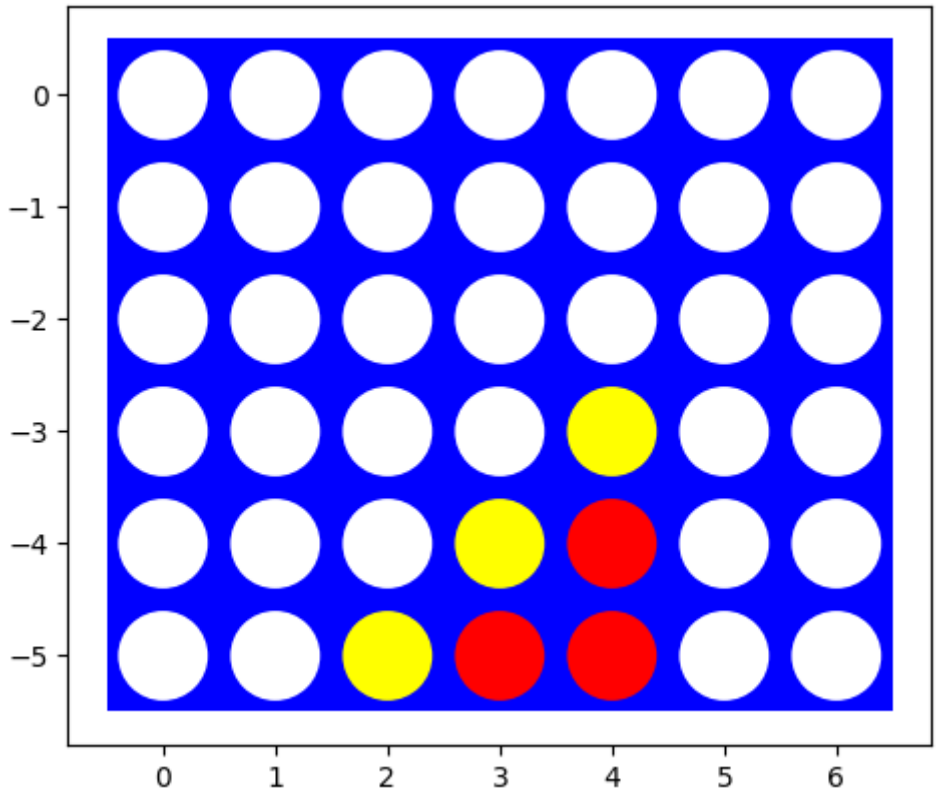

```
Enter the column:4
```



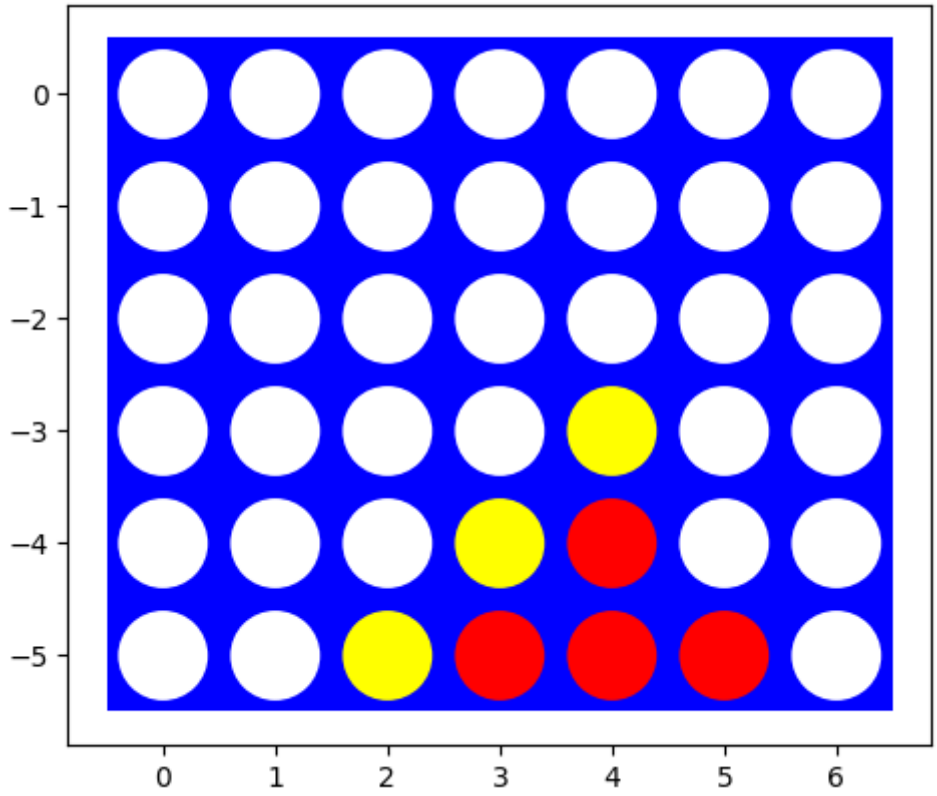


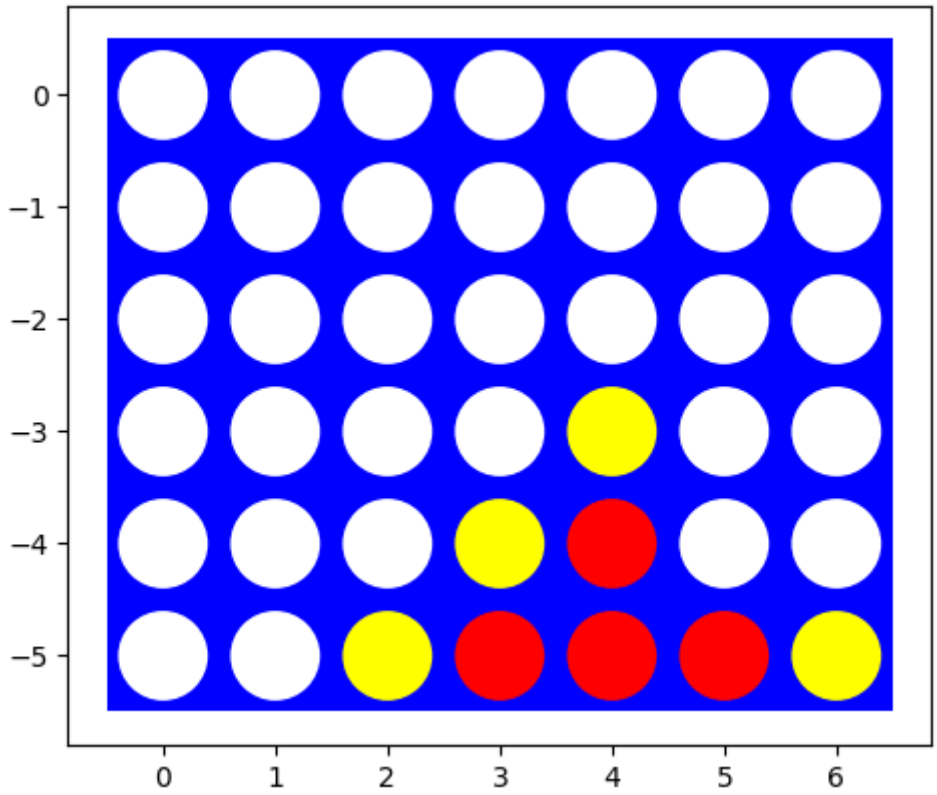

```
Enter the column:4
```



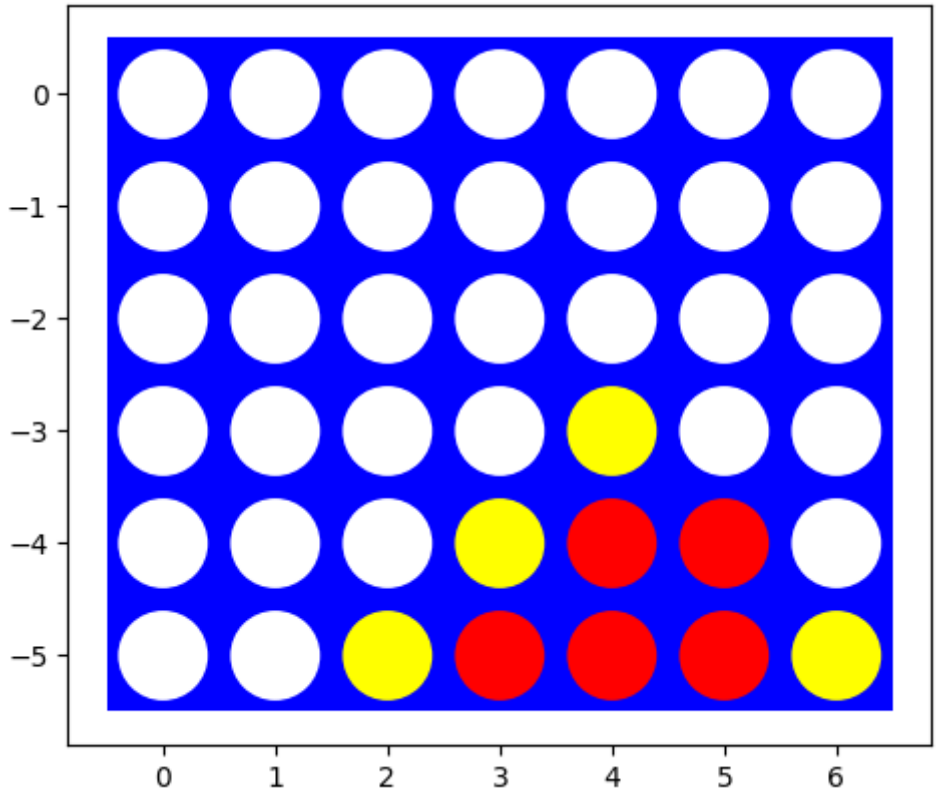


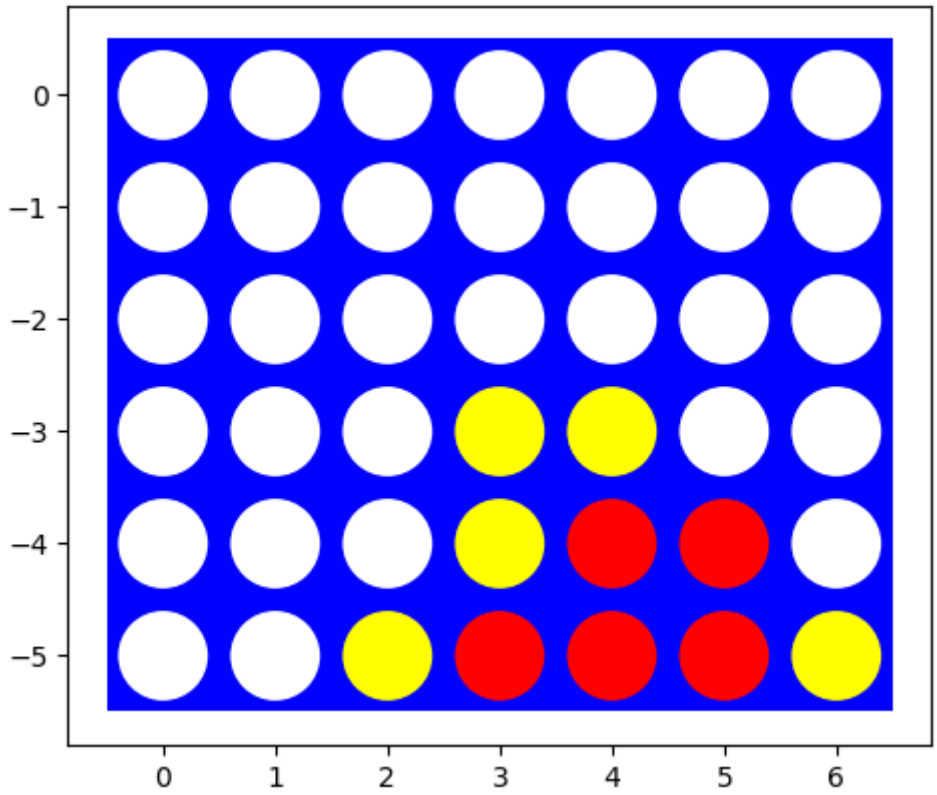

```
Enter the column:5
```



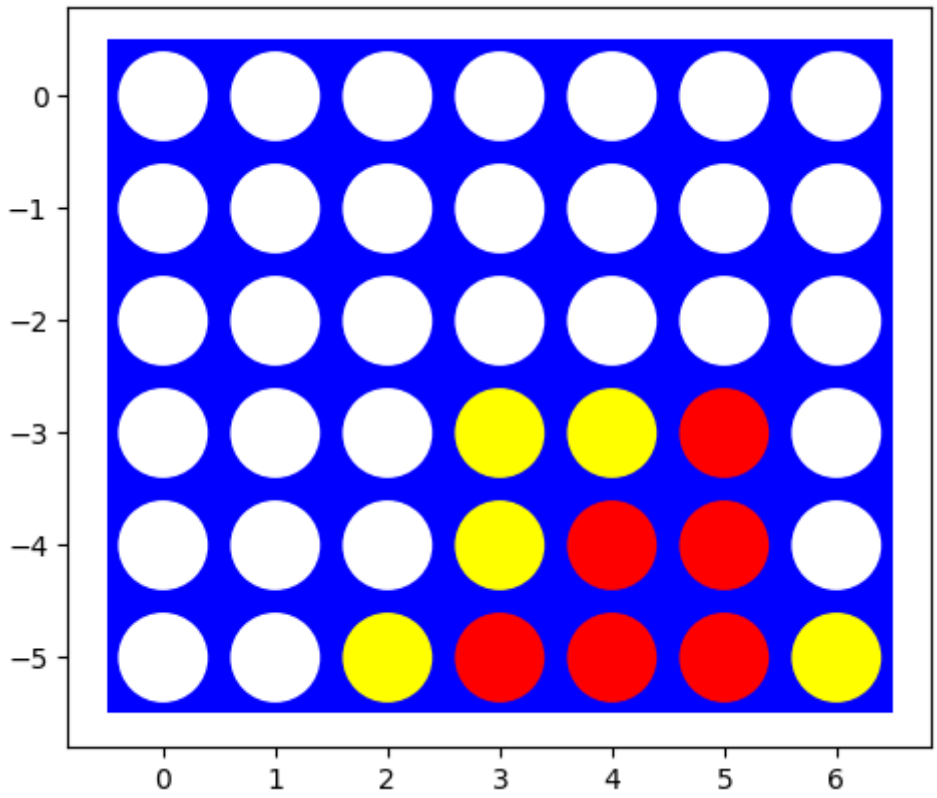



```
Enter the column:5
```

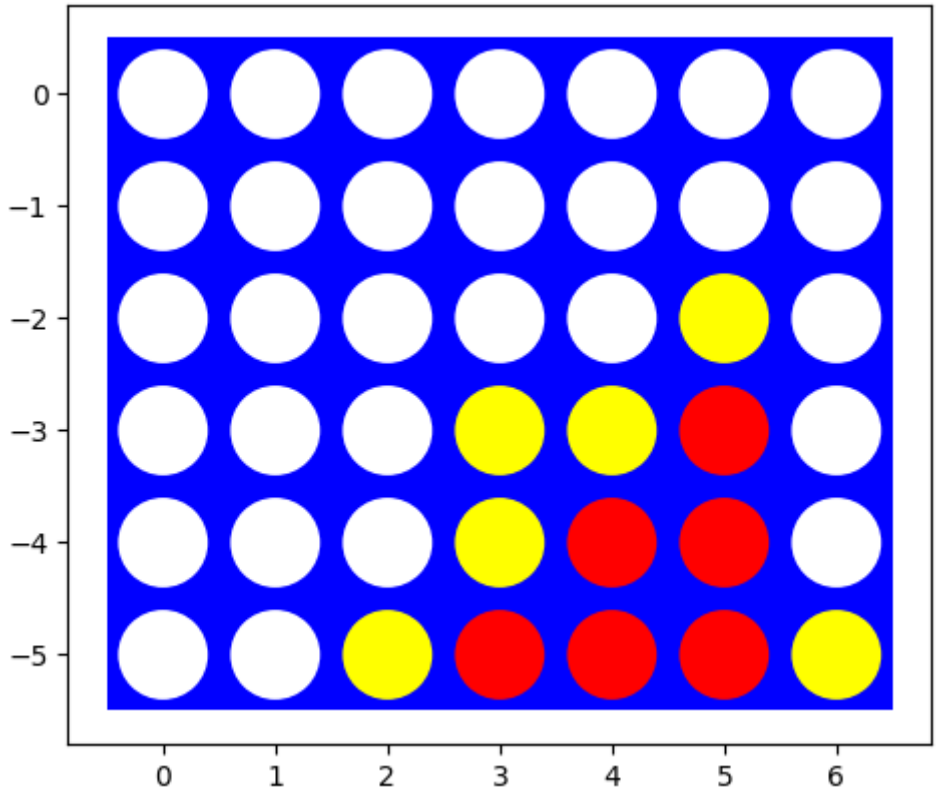





```
Enter the column:5
```




```
AI WIN!
```



How long does it take to make a move? Start with a smaller board with 4 columns and make the board larger by adding columns.

In [15]:

```
import time

def measure_time_for_move(board, player, depth):
    start = time.time()
    _, _ = minmax(board, player, -math.inf, math.inf, depth)
    return time.time() - start
```

```
def main():
    for num_columns in range(4, 8):  # You can adjust the range based on the desired
        board = initialize_board(6, num_columns)
        #visualize(board)

        # Time measurement for Human move
        human_time = measure_time_for_move(board, 1, 1)
        print(f"Time for Human move on a {6}x{num_columns} board: {human_time:.5f} s

        # Time measurement for AI move
        ai_time = measure_time_for_move(board, 2, 3)  # Adjust depth based on your c
        print(f"Time for AI move on a {6}x{num_columns} board: {ai_time:.5f} seconds

if __name__ == "__main__":
    main()
```

```
Time for Human move on a 6x4 board: 0.00292 seconds
Time for AI move on a 6x4 board: 0.03906 seconds
Time for Human move on a 6x5 board: 0.00160 seconds
Time for AI move on a 6x5 board: 0.03892 seconds
Time for Human move on a 6x6 board: 0.00151 seconds
Time for AI move on a 6x6 board: 0.04700 seconds
Time for Human move on a 6x7 board: 0.00157 seconds
Time for AI move on a 6x7 board: 0.03887 seconds
```

Move ordering [5 points]

Starting the search with better moves will increase the efficiency of alpha-beta pruning. Describe and implement a simple move ordering strategy. Make a table that shows how the ordering strategies influence the time it takes to make a move?

Description: The code defines functions for ordering moves (order_moves), obtaining the board after a move (board_after_move), and implementing Minimax with move ordering (minmax_with_move_ordering). Move ordering is done by evaluating and sorting the valid moves based on a heuristic function, which is defined using the heuristic function (assumed to be defined elsewhere). The minmax_with_move_ordering function is an enhanced version of the standard Minimax algorithm. It orders the moves based on the heuristic before exploring them, potentially improving the efficiency of the algorithm. The code includes a time measurement experiment to compare the performance of Minimax with and without move ordering.

In [46]:

```
import copy
import random
import math
import time

def order_moves(board, maxplayer):
    valid_moves = valid_actions(board)
    return sorted(valid_moves, key=lambda move: heuristic(board_after_move(board, mo

def board_after_move(board, move, player):
    new_board = copy.deepcopy(board)
    row = get_row(new_board, move)
    update_board(new_board, row, move, player)
    return new_board

def minmax_with_move_ordering(board, maxplayer, alpha, beta, depth):
    terminal_node = is_terminal_state(board)
    if depth == 0 or terminal_node:
        if terminal_node:
            if check_winner(board, 2):
```

```
                return (1, None)
            elif check_winner(board, 1):
                return (-1, None)
            else:
                return (0, None)
        elif depth == 0:
            return (evaluate(board, 2), None)
    elif maxplayer:
        valid_moves = order_moves(board, maxplayer)
        priority = -math.inf
        prior_col = random.choice(valid_moves)
        for poss_col in valid_moves:
            new_board = board_after_move(board, poss_col, 2)
            updated_prior = minmax_with_move_ordering(new_board, False, alpha, beta,
            if priority < updated_prior:
                priority = updated_prior
                prior_col = poss_col
            alpha = max(alpha, priority)
            if alpha >= beta:
                break

        return priority, prior_col
    else:
        valid_moves = order_moves(board, not maxplayer)
        prior_col = random.choice(valid_moves)
        priority = math.inf
        for poss_col in valid_moves:
            new_board = board_after_move(board, poss_col, 1)
            updated_prior = minmax_with_move_ordering(new_board, True, alpha, beta,
            if priority > updated_prior:
                prior_col = poss_col
                priority = updated_prior
            beta = min(beta, priority)
            if alpha >= beta:
                break
        return priority, prior_col

# Measure time for different move ordering strategies
if __name__ == "__main__":
    rows, cols = 6, 7
    board = initialize_board(rows, cols)

    # Time measurement without move ordering
    start_time = time.time()
    ai_move = minmax(board, 2, -math.inf, math.inf, 3)[1]
    end_time = time.time()
    print(f"Time without move ordering: {end_time - start_time} seconds")

    # Time measurement with move ordering
    start_time = time.time()
    ai_move_ordered = minmax_with_move_ordering(board, 2, -math.inf, math.inf, 3)[1]
    end_time = time.time()
    print(f"Time with move ordering: {end_time - start_time} seconds")
```

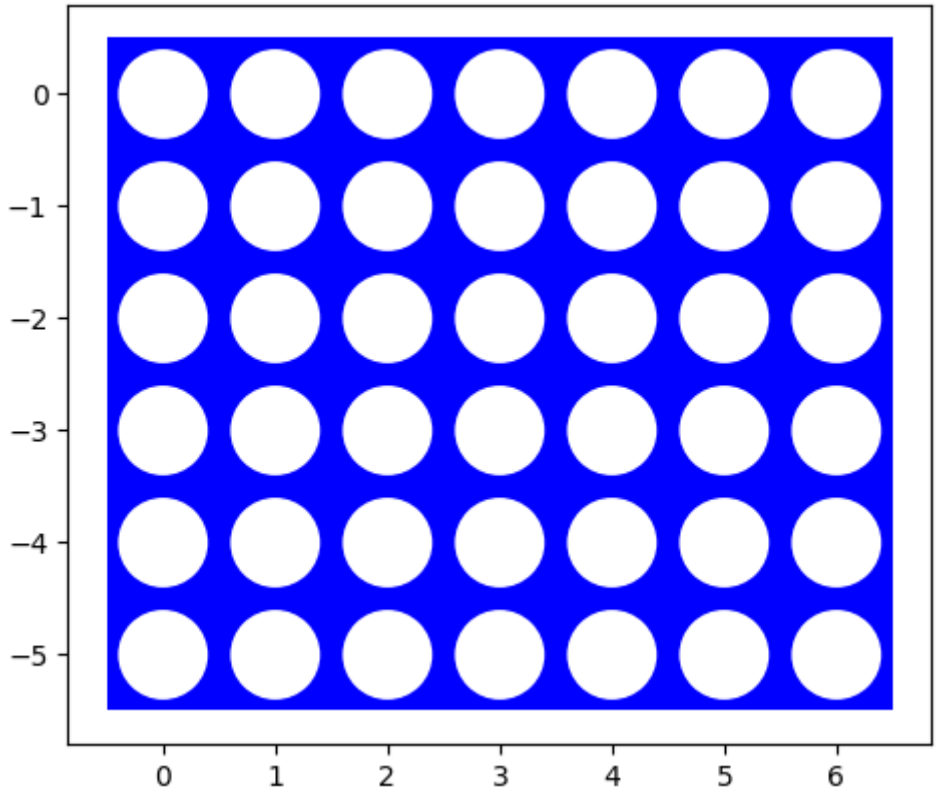
```
Time without move ordering: 0.030957698822021484 seconds
Time with move ordering: 0.017600059509277344 seconds
```

The first few moves [5 points]

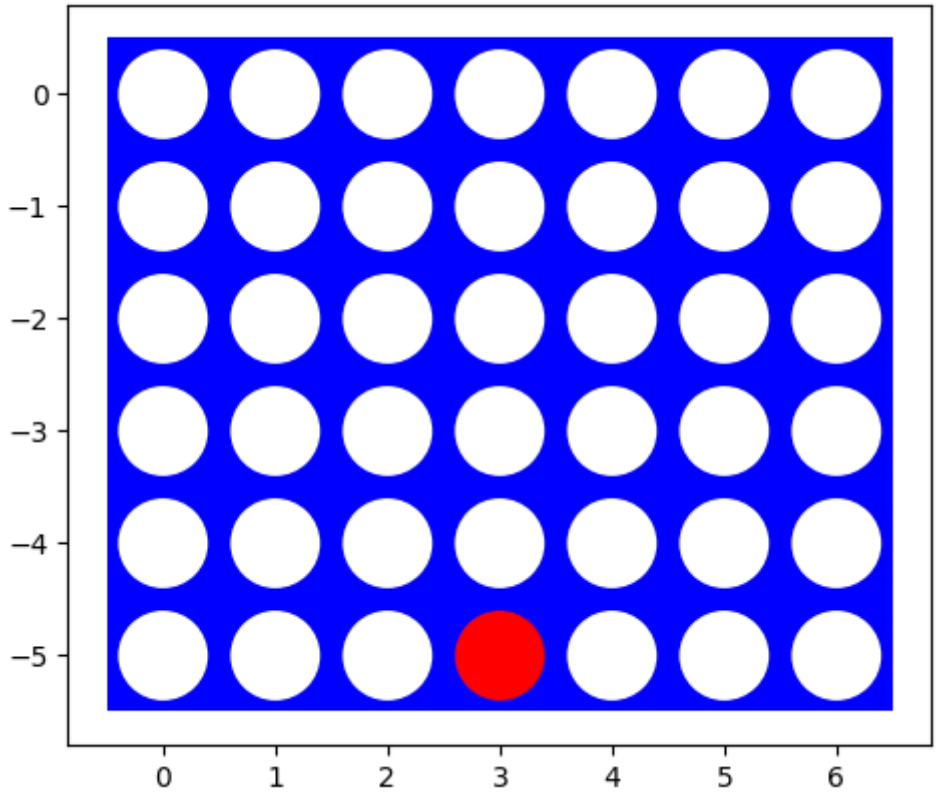
Start with an empty board. This is the worst case scenario for minimax search since it needs solve all possible games that can be played (minus some pruning) before making the decision. What can you do?

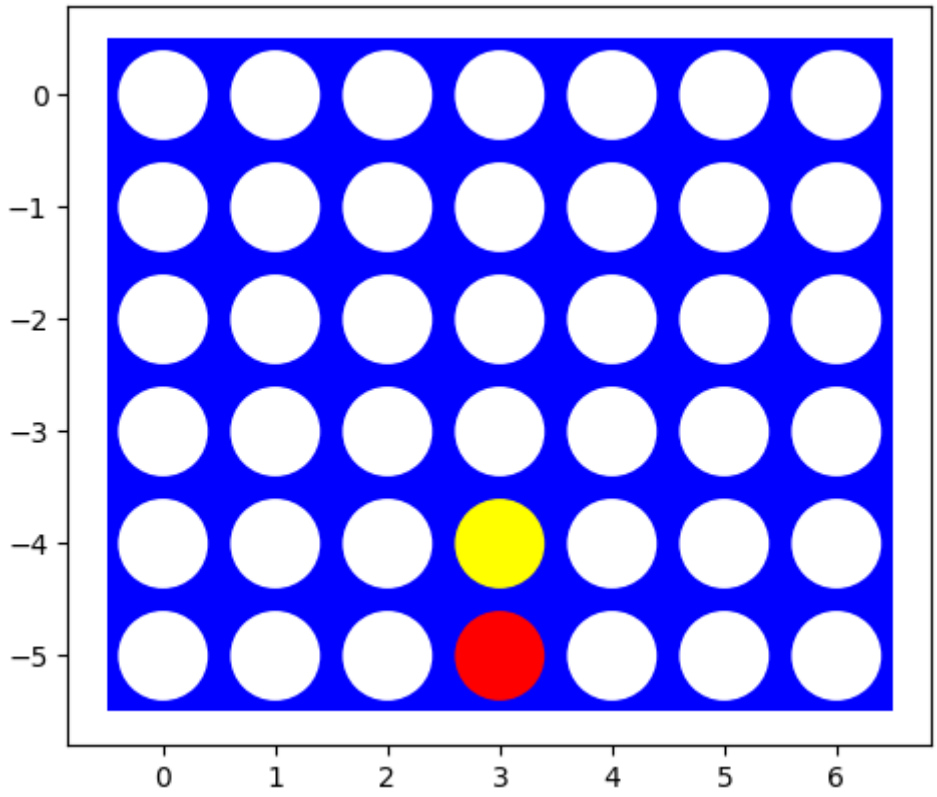
In [48]:

```
# Your code/ answer goes here.
game_over_check = False
turn = 0
difficulty = 0


board = initialize_board(6,7)
visualize(board)

while not game_over_check:
    if turn == 0:
        col = int(input("Enter the column:"))
        if valid_actions_check(board, col):
            row = get_row(board, col)
            update_board(board, row, col, 1)
            #print_board(board)
            if check_winner(board,1):
                print("YOU WIN !")
                game_over_check = True
            turn += 1
            visualize(board)
    elif turn == 1:
        func_score, best_col = minmax(board, True, -math.inf, math.inf, 3)
        if valid_actions_check(board, best_col):
            row = get_row(board, best_col)
            update_board(board, row, best_col, 2)
            #print_board(board)
            if check_winner(board, 2):
                print("AI WIN!")
                game_over_check = True
            turn -= 1
            visualize(board)
```

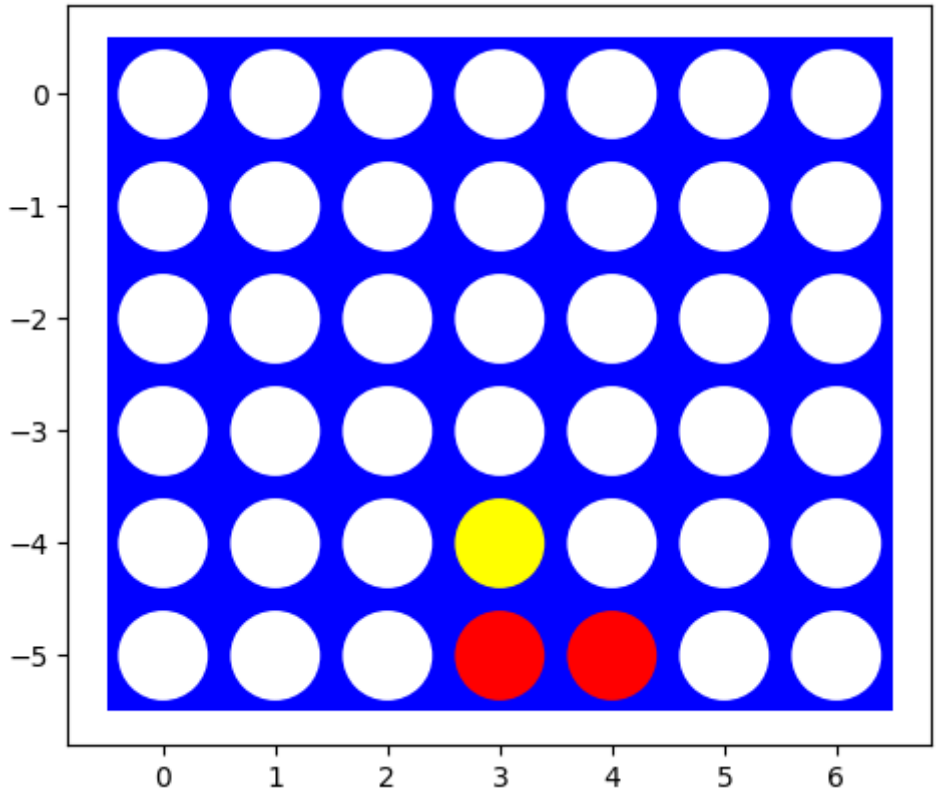


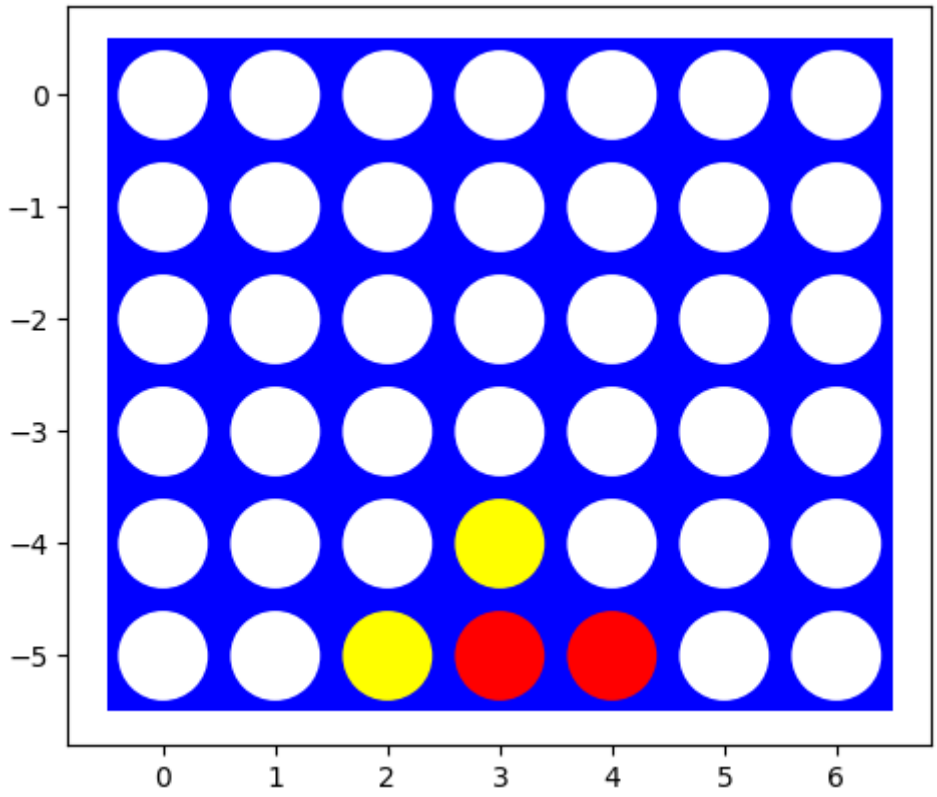

```
Enter the column:3
```



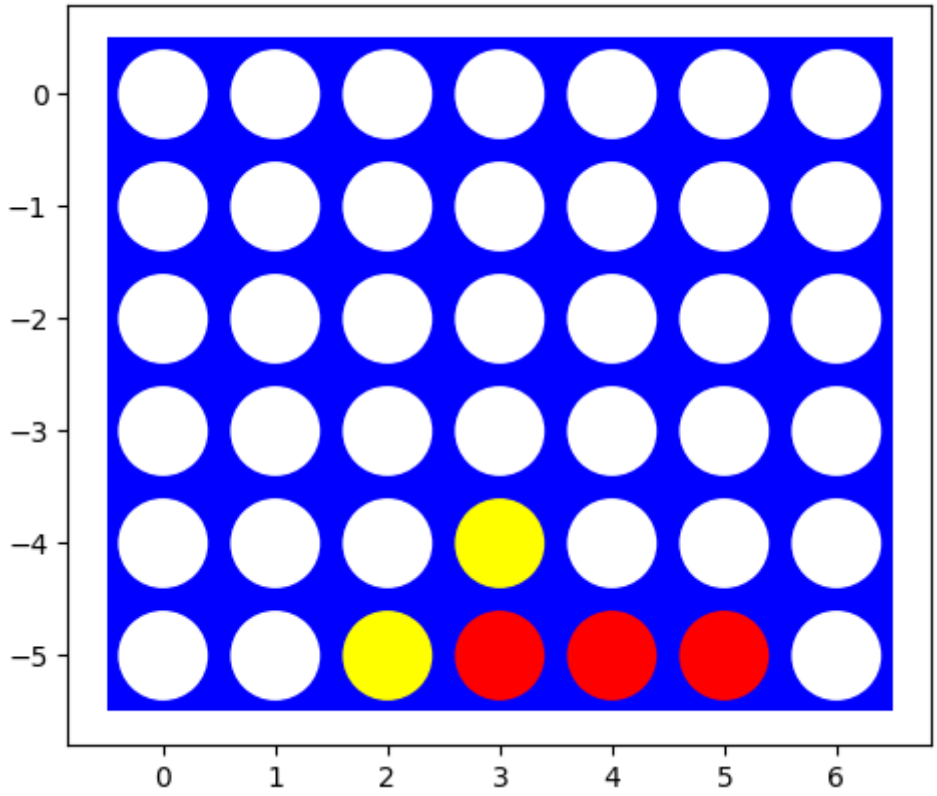


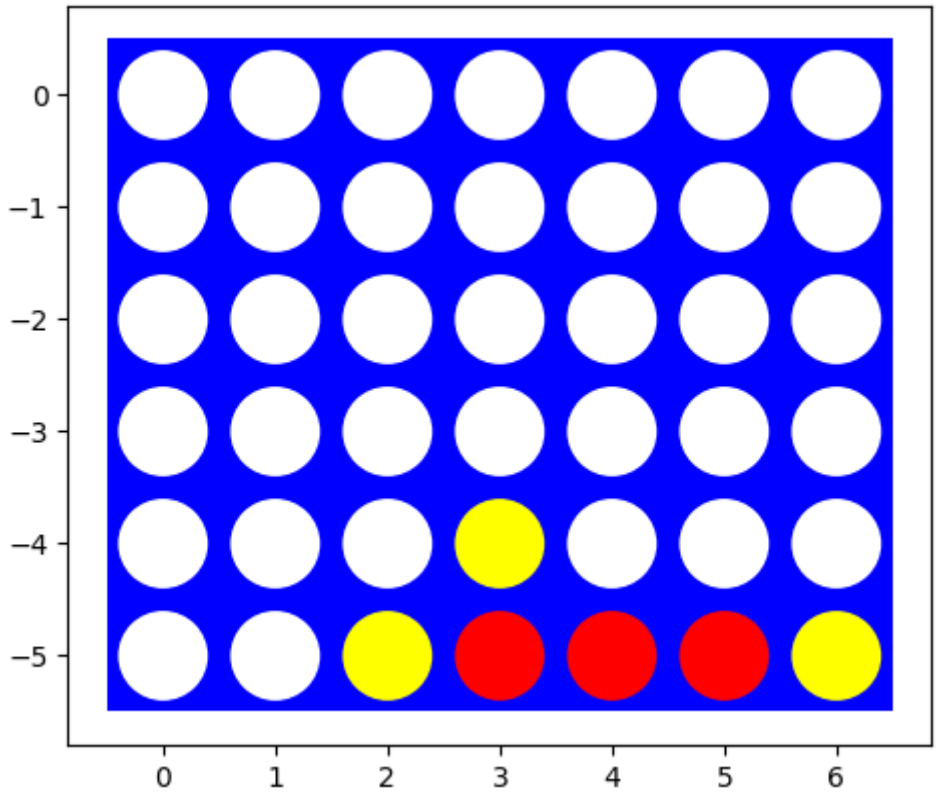

```
Enter the column:4
```



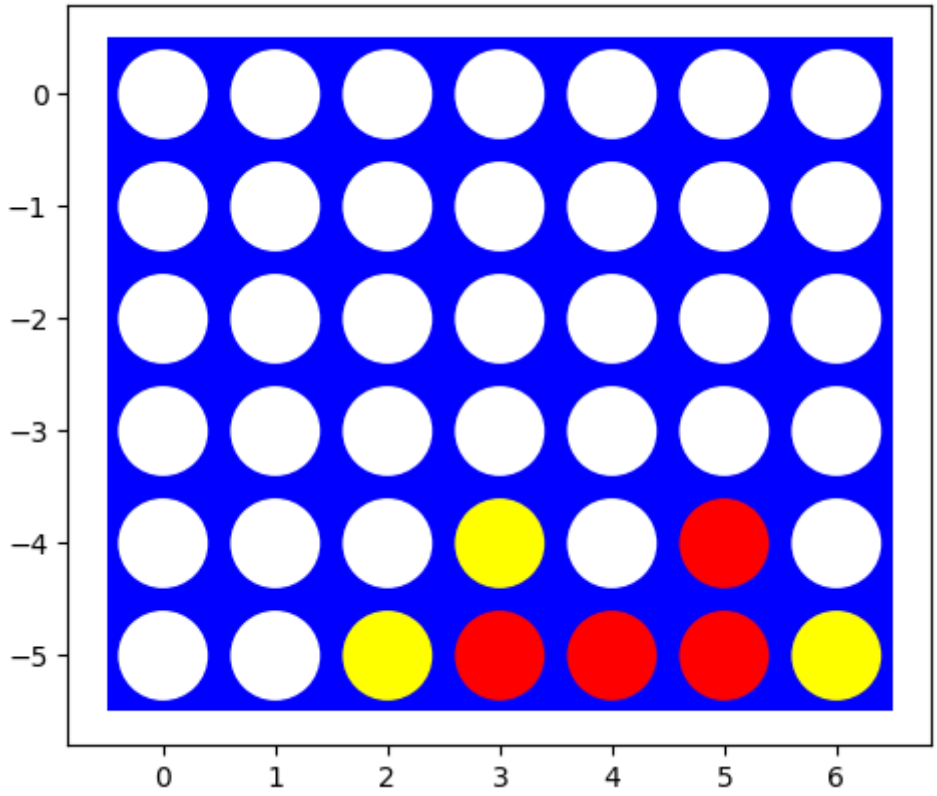


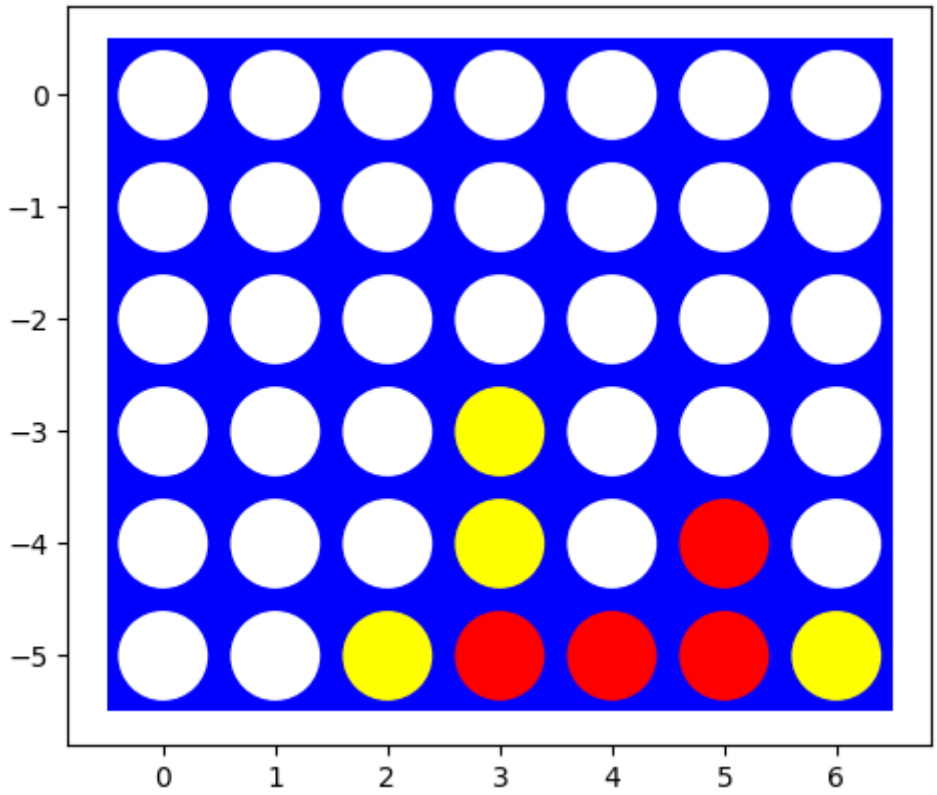

```
Enter the column:5
```



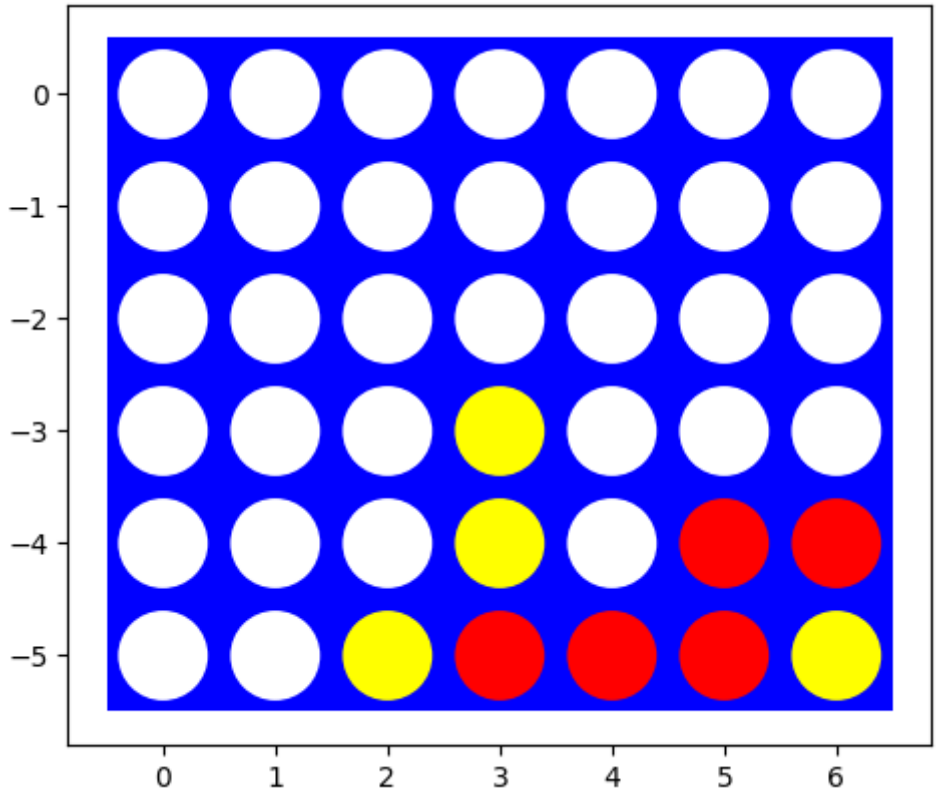


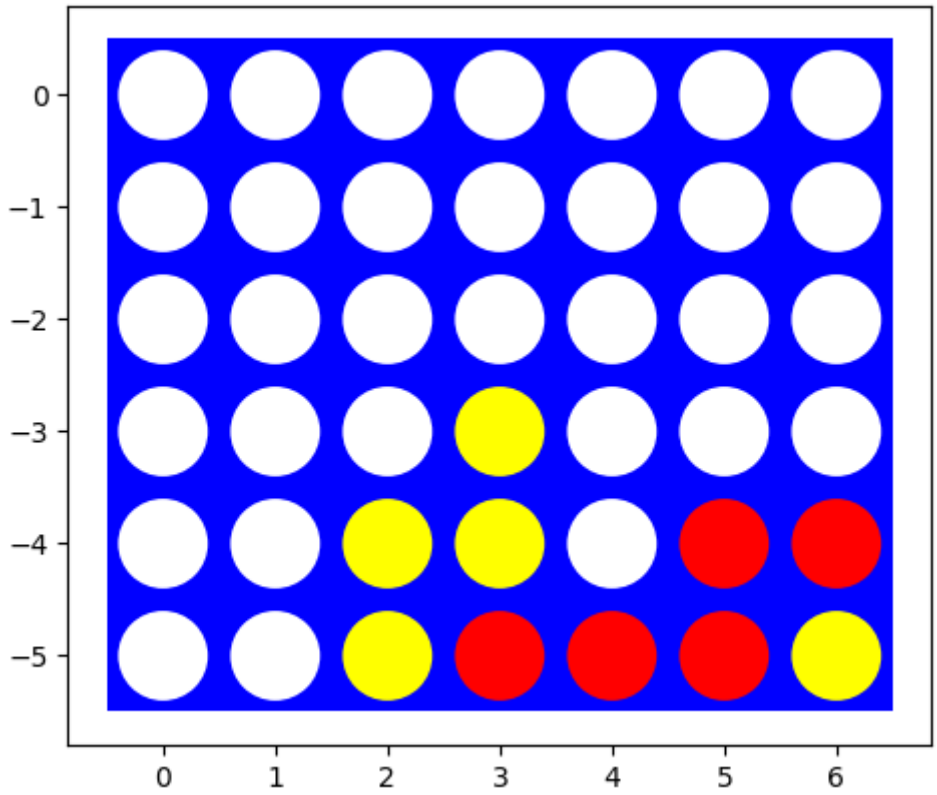

```
Enter the column:5
```



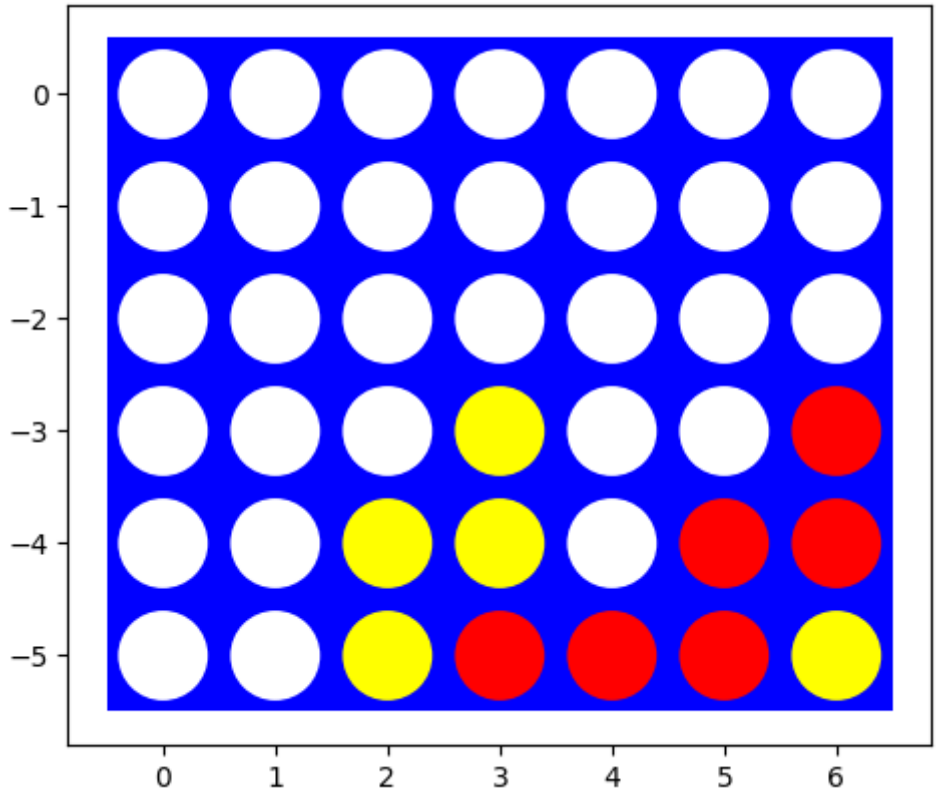


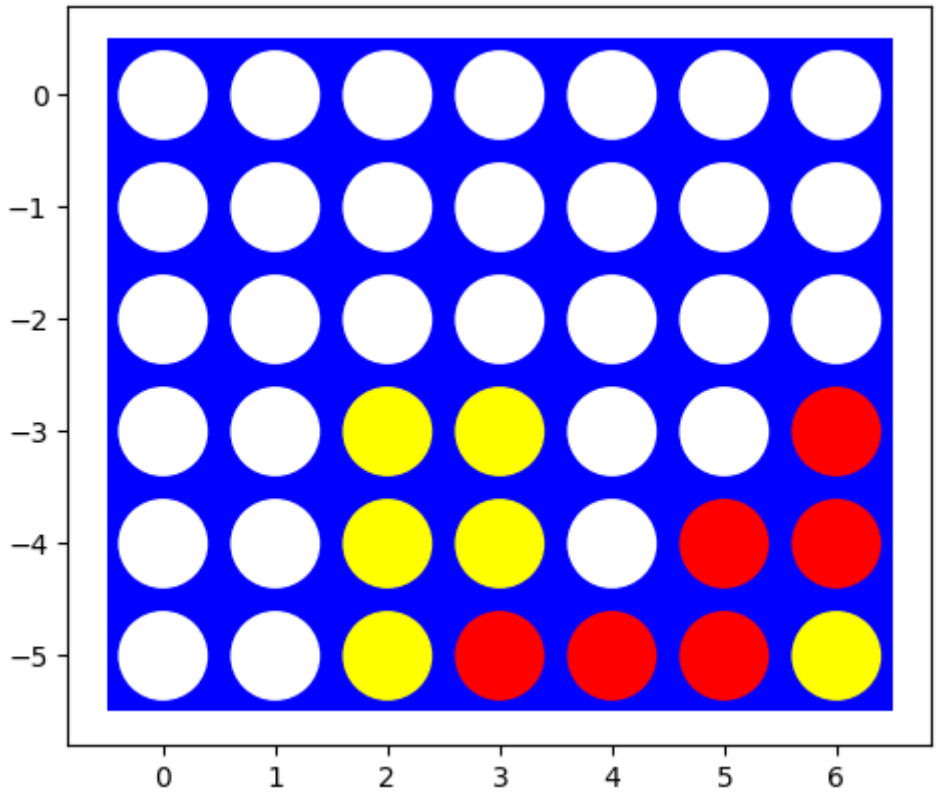

```
Enter the column:6
```



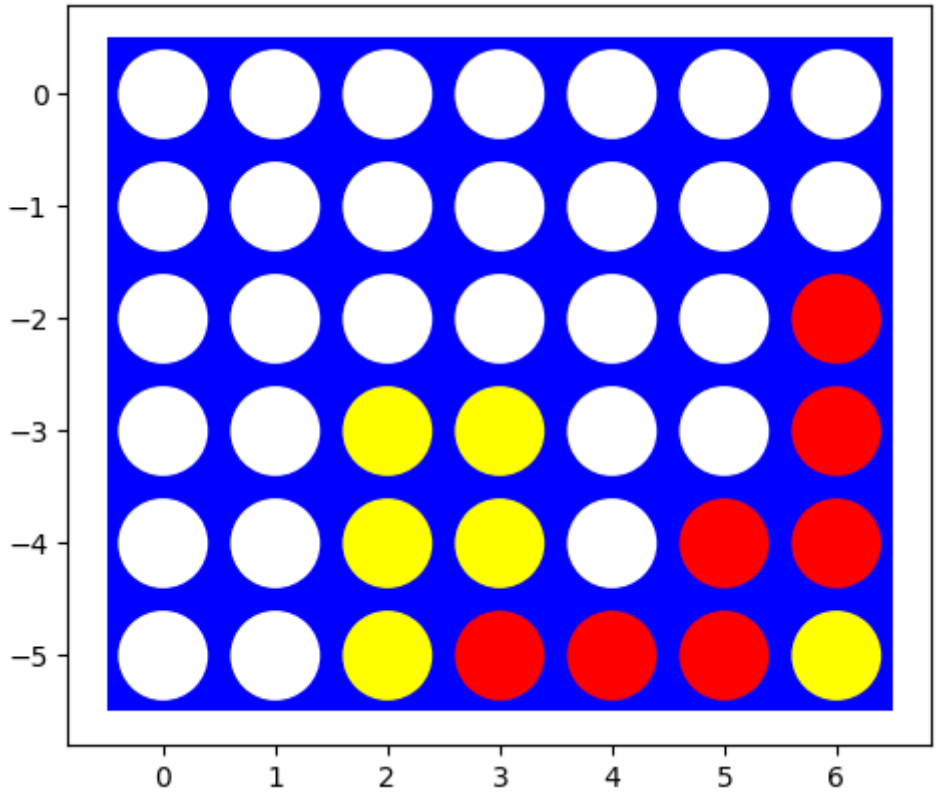


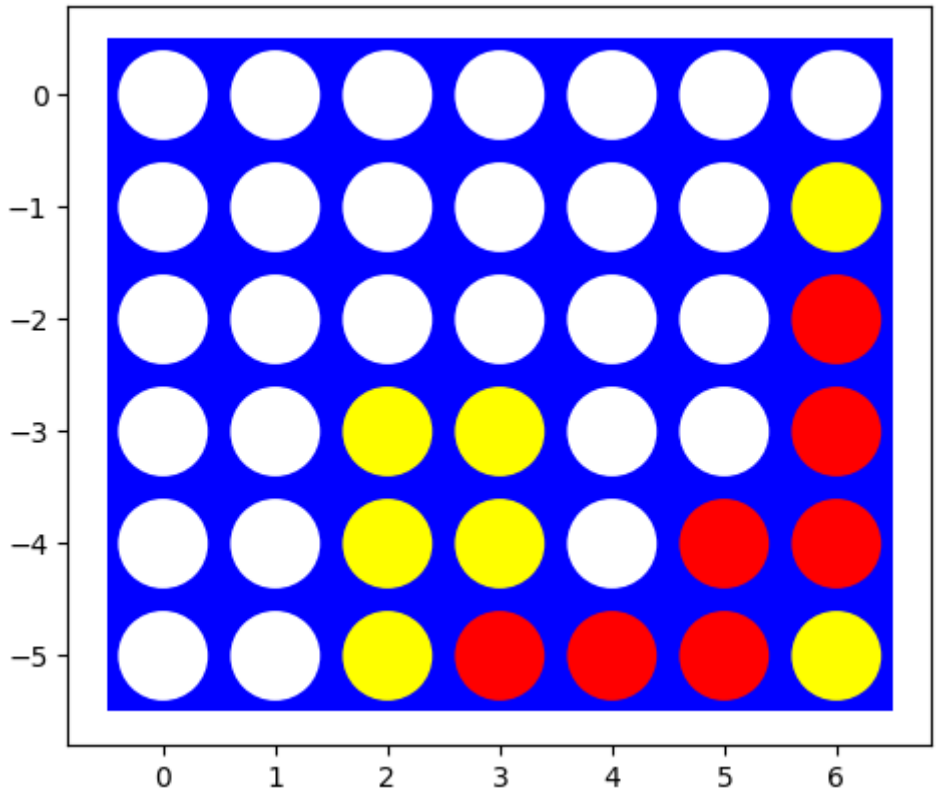

```
Enter the column:6
```



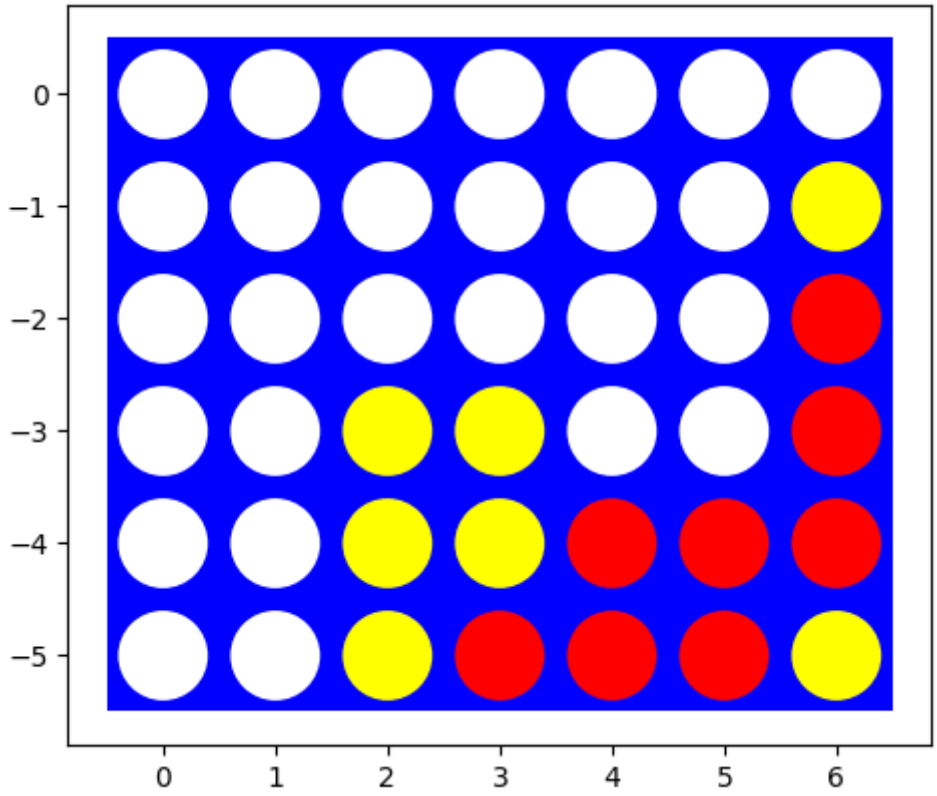


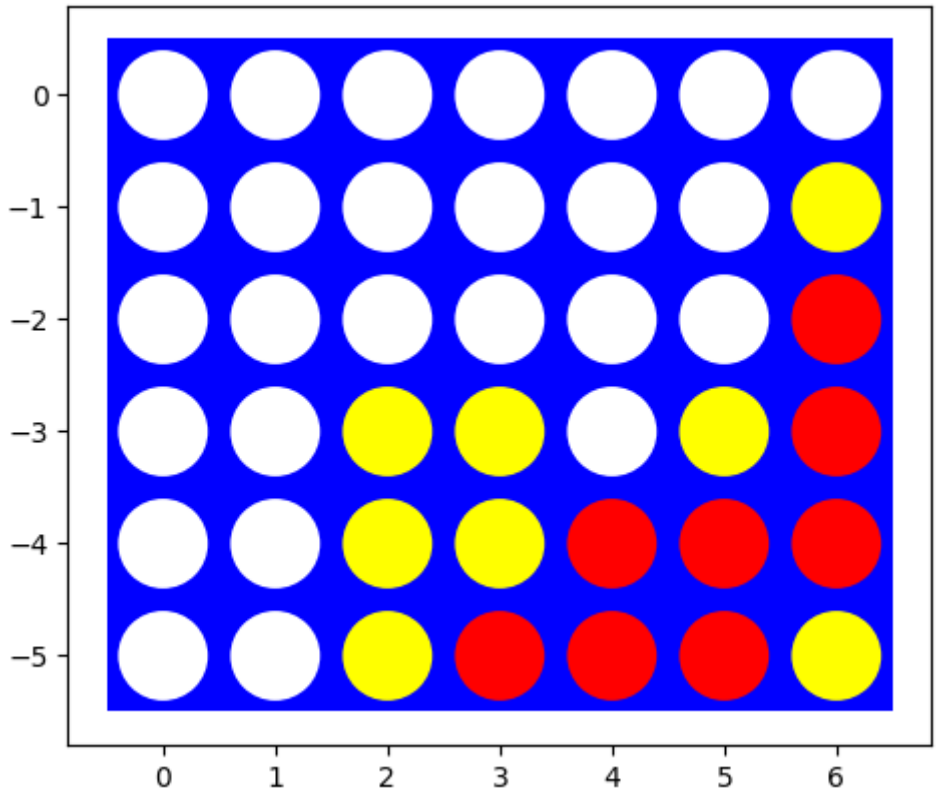

```
Enter the column:6
```



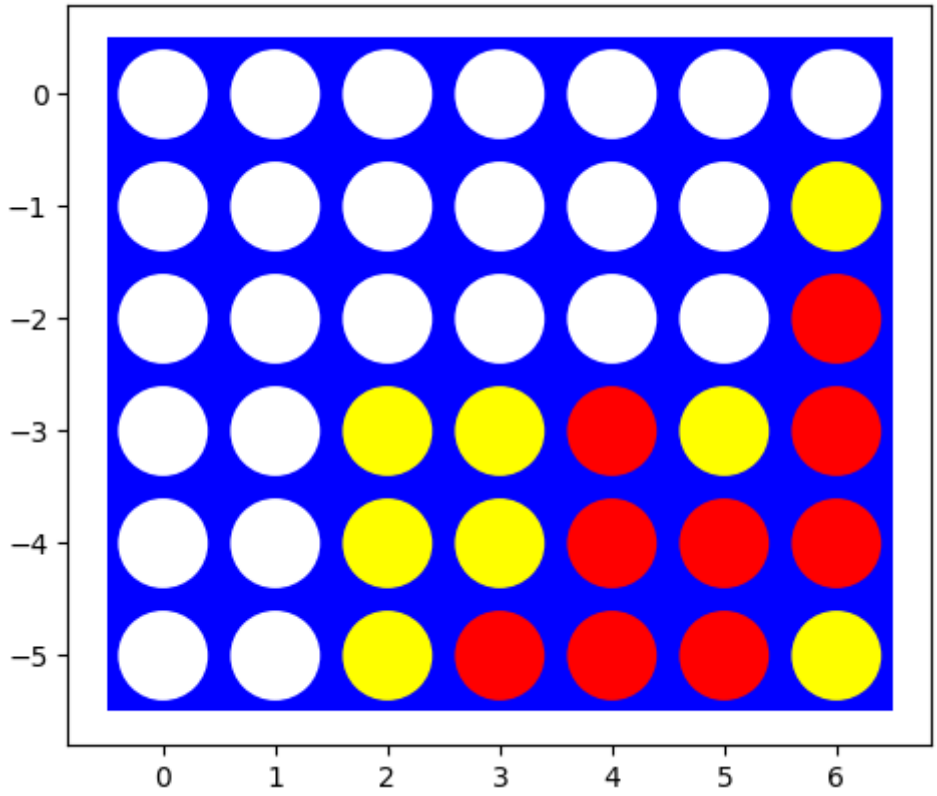


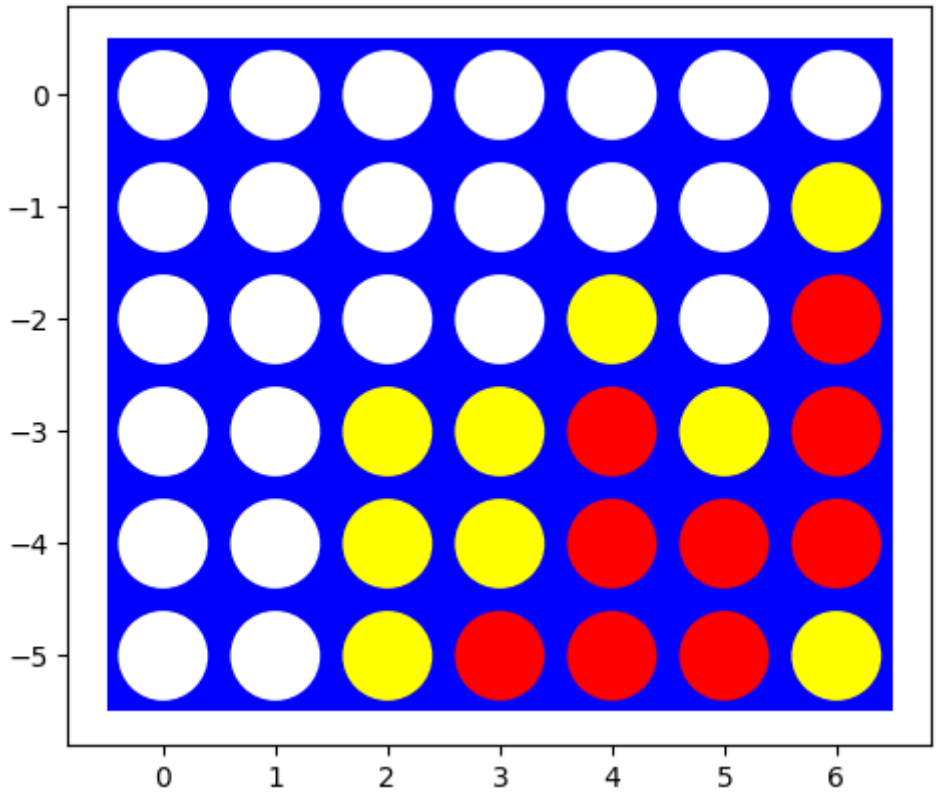

```
Enter the column:4
```



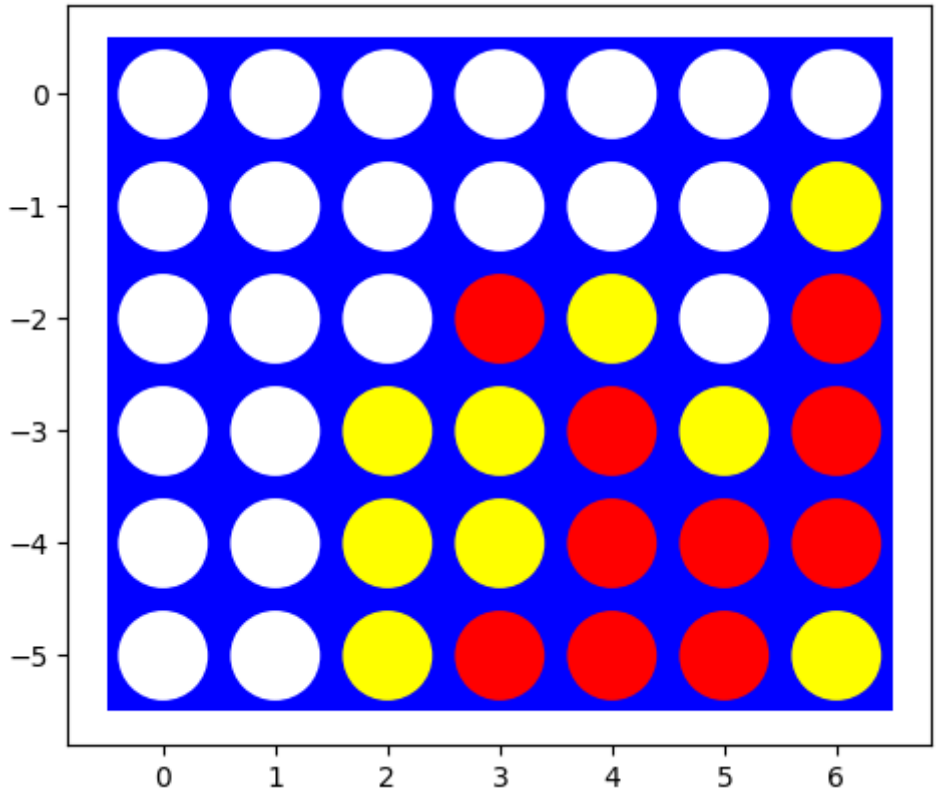


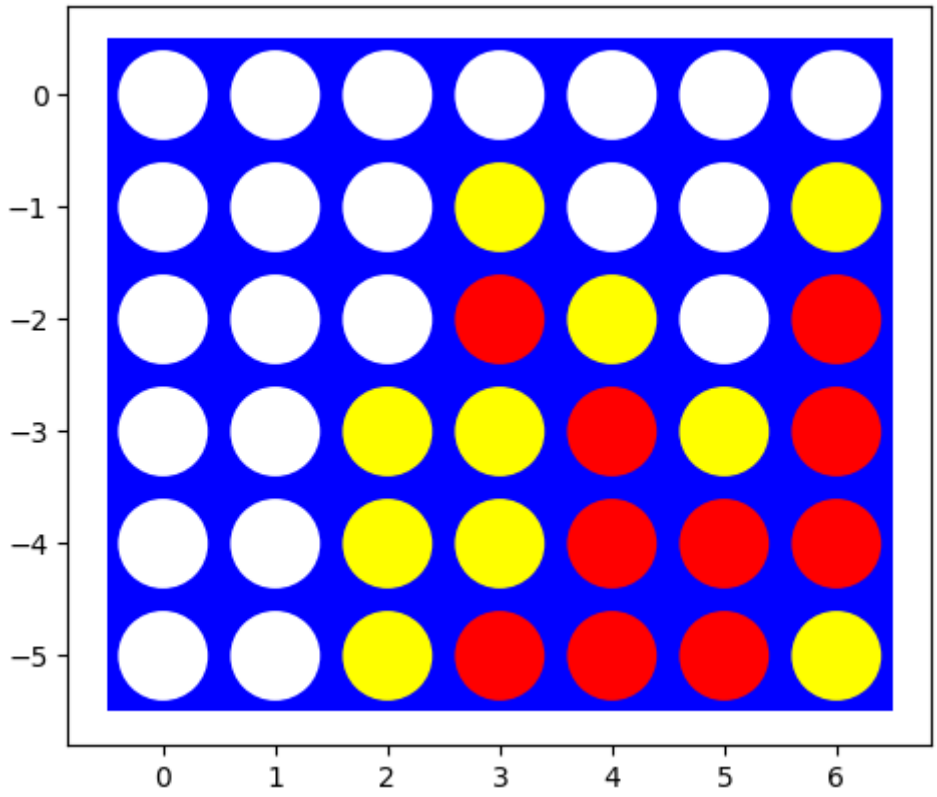

```
Enter the column:4
```



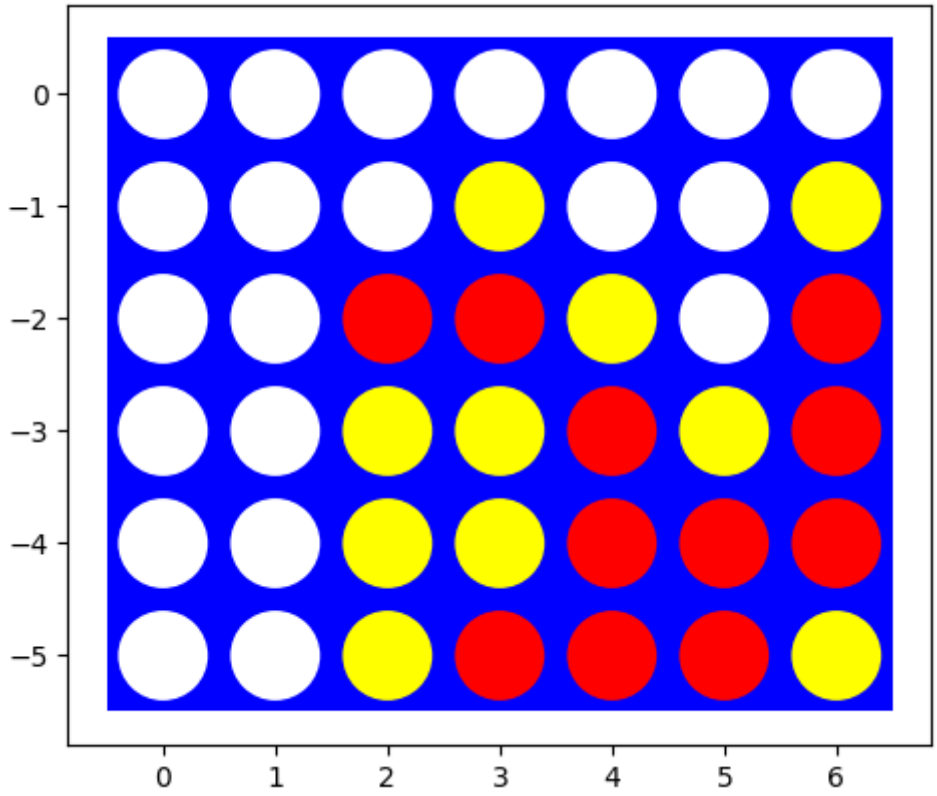


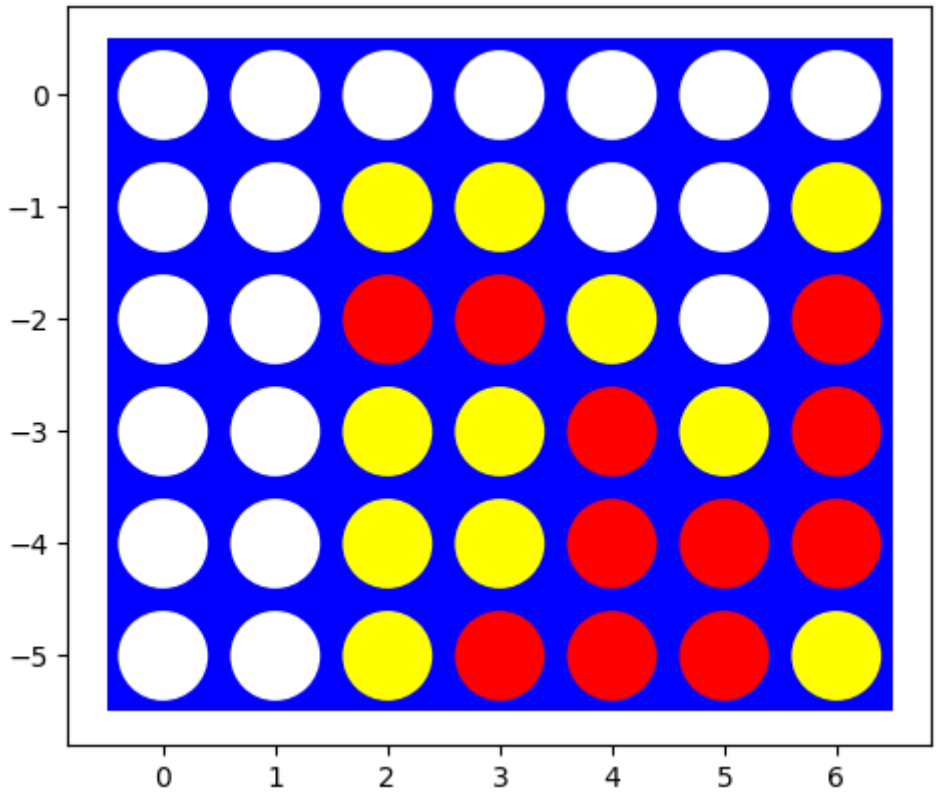

```
Enter the column:3
```



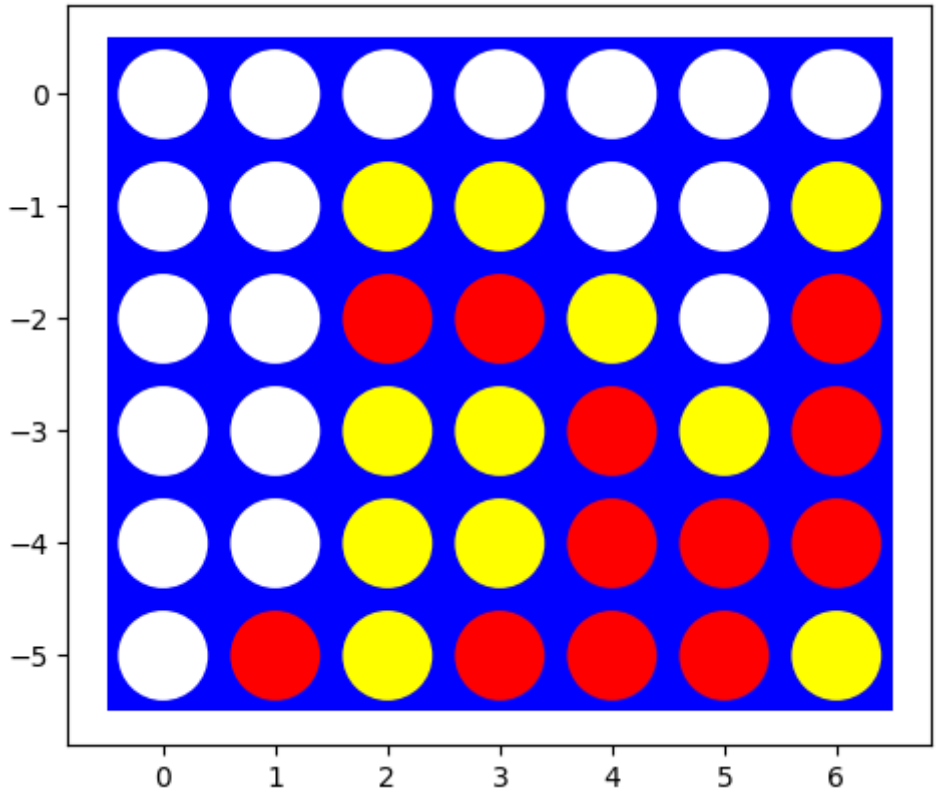


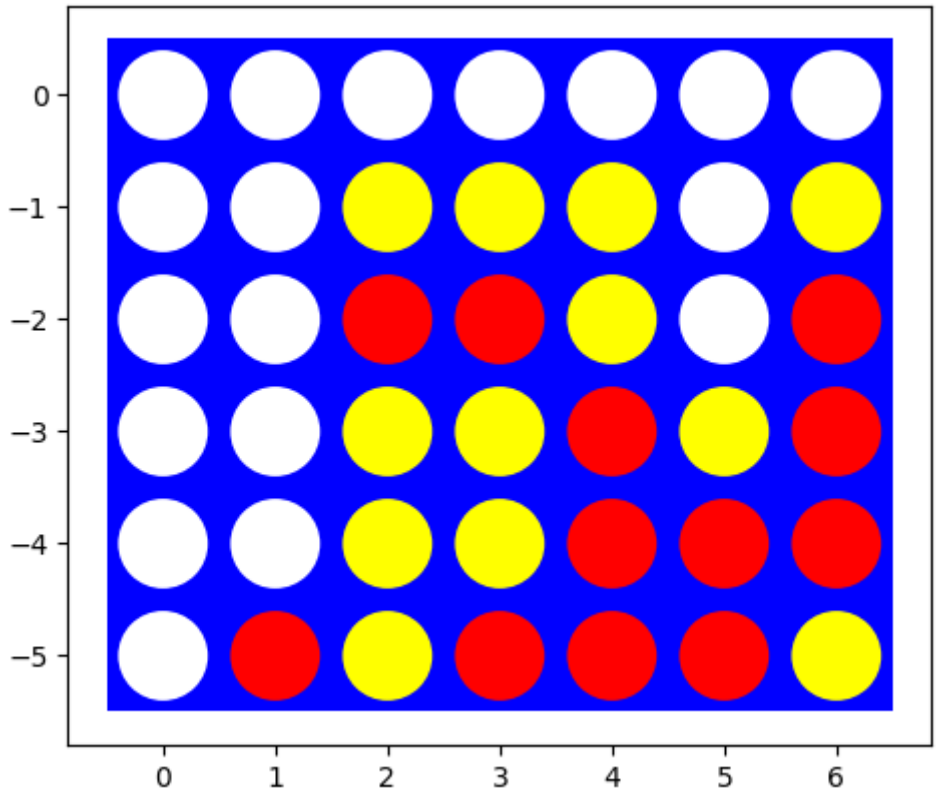

```
Enter the column:2
```



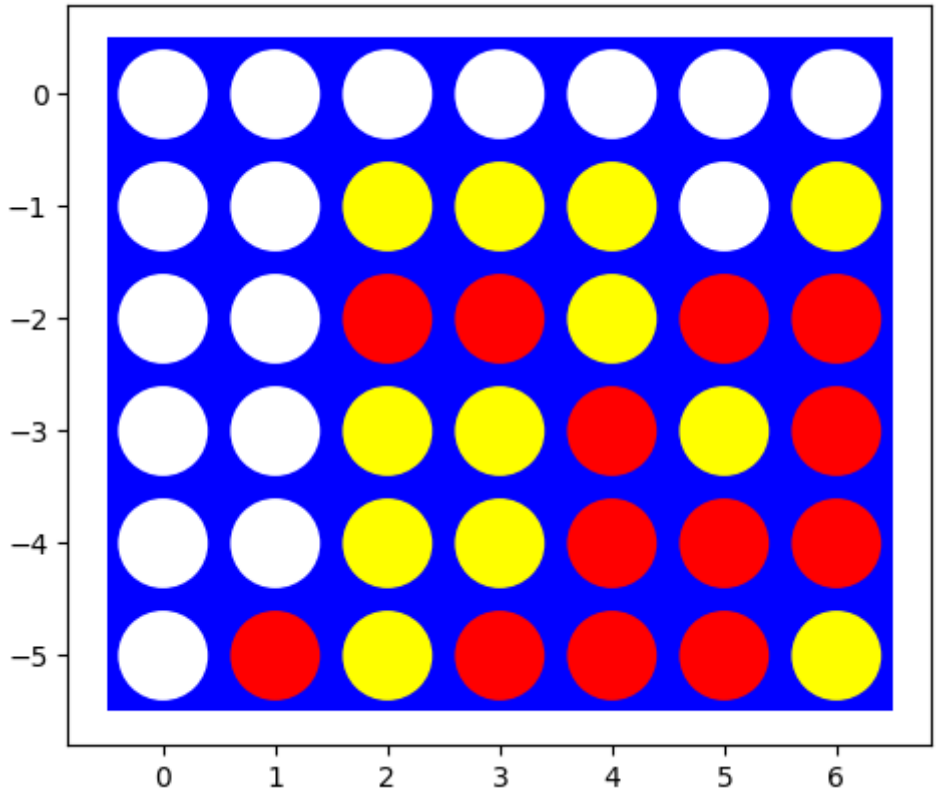


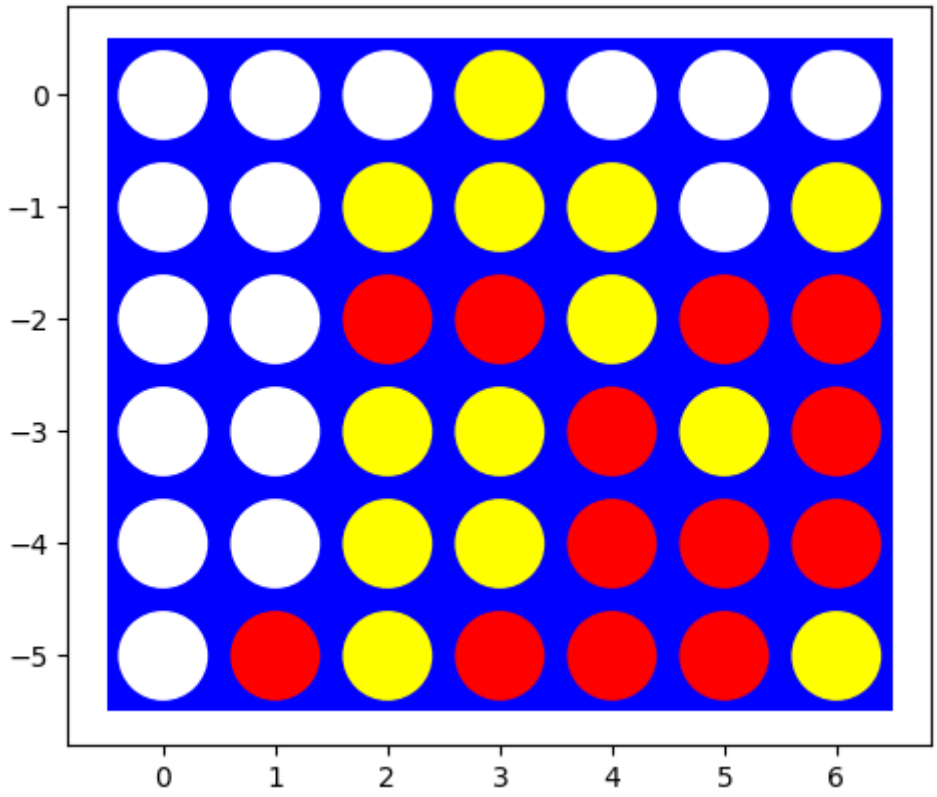

```
Enter the column:1
```



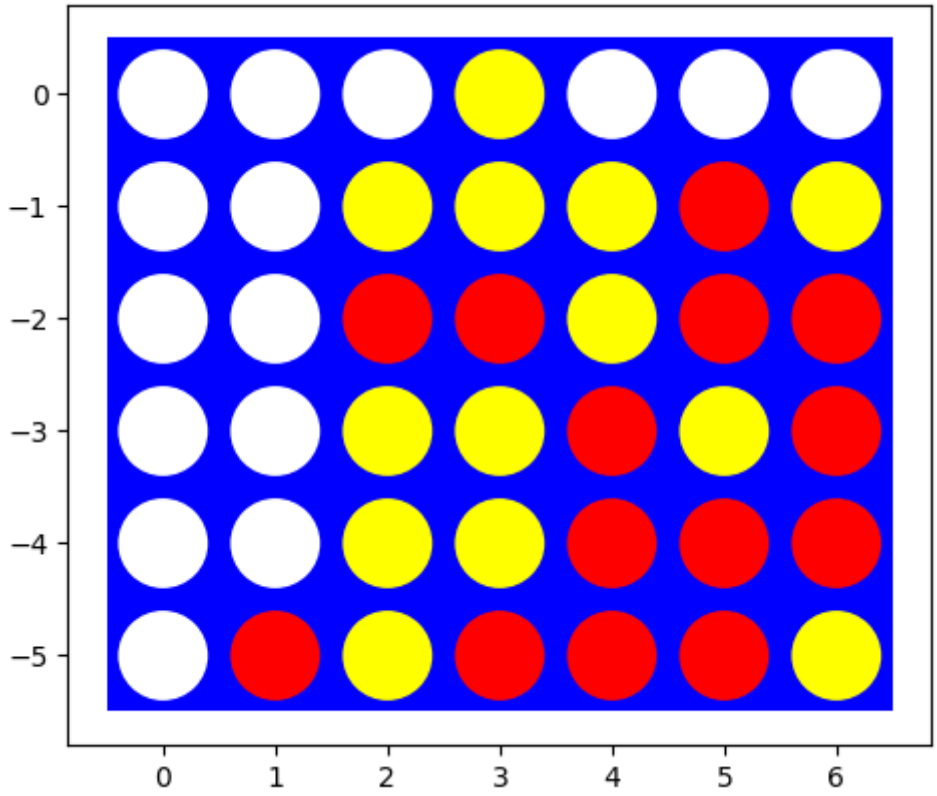


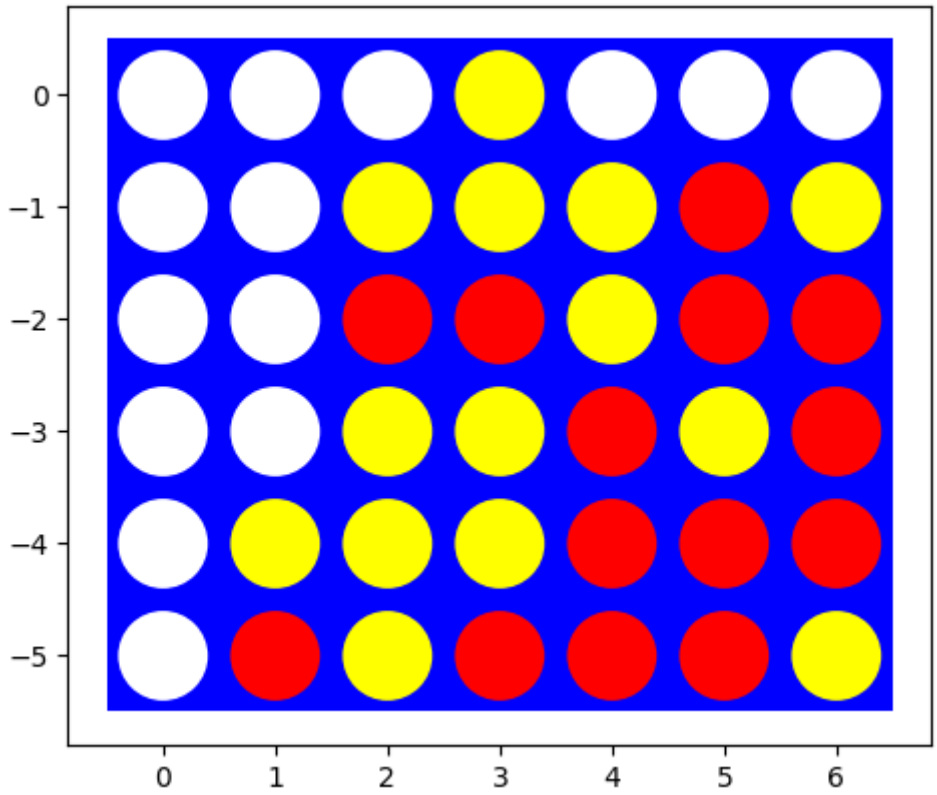

```
Enter the column:5
```



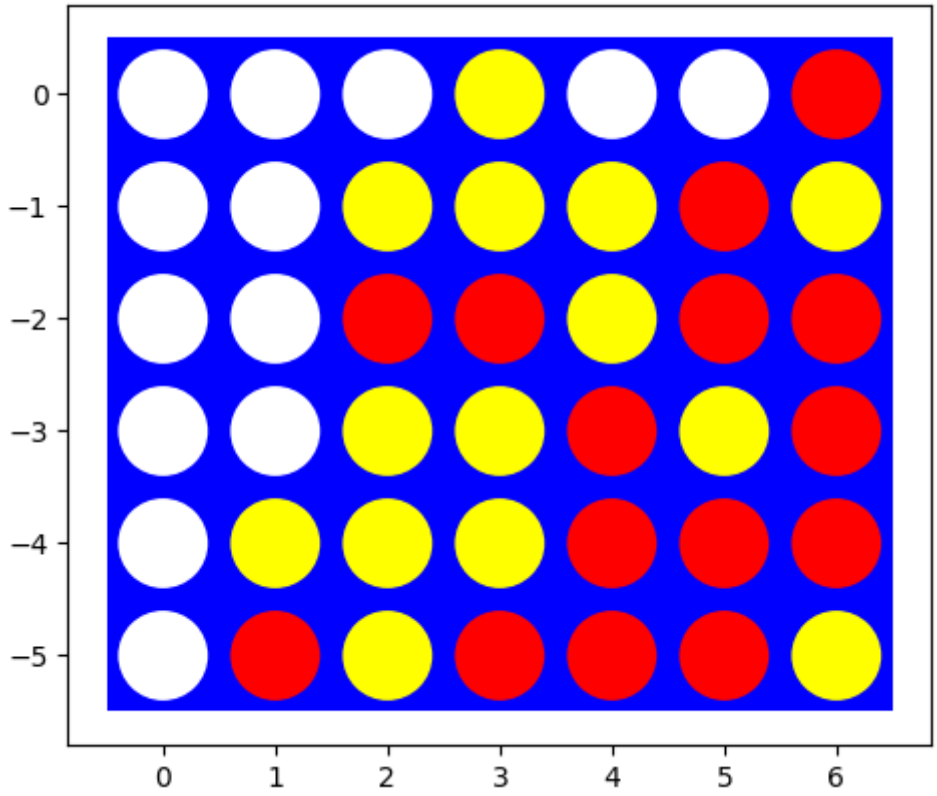


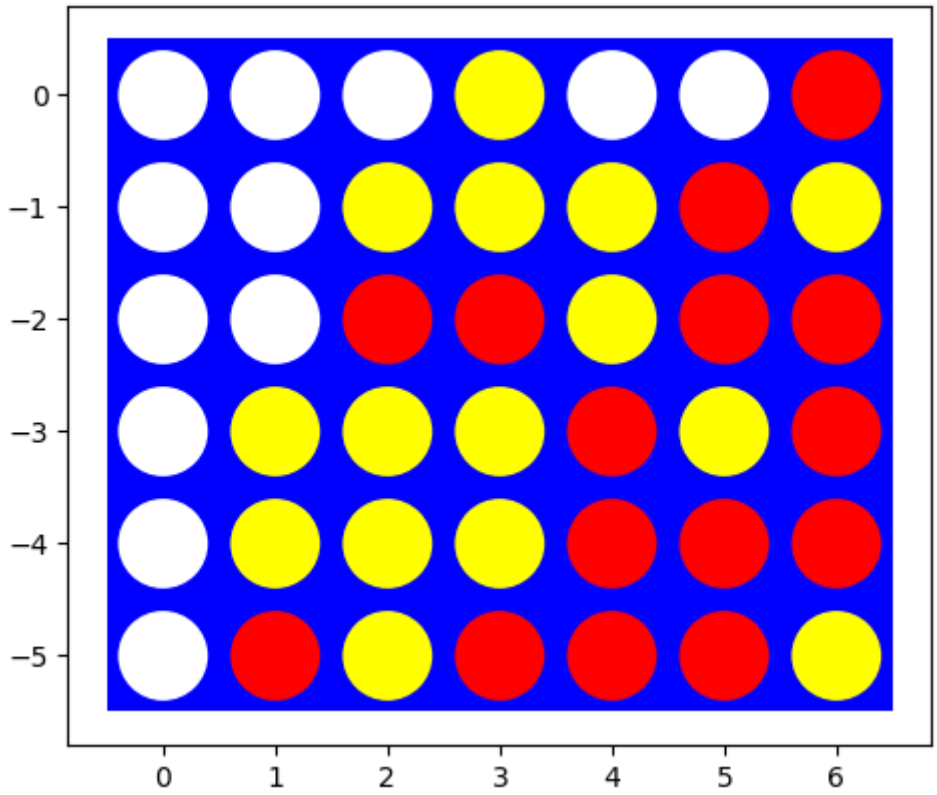

```
Enter the column:5
```



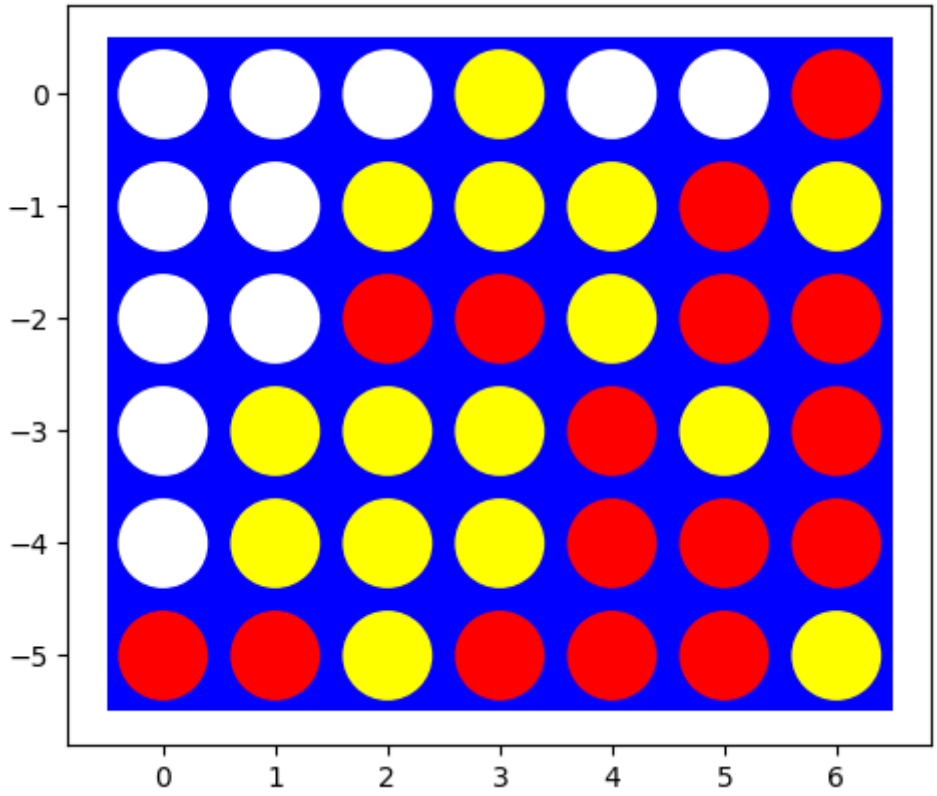


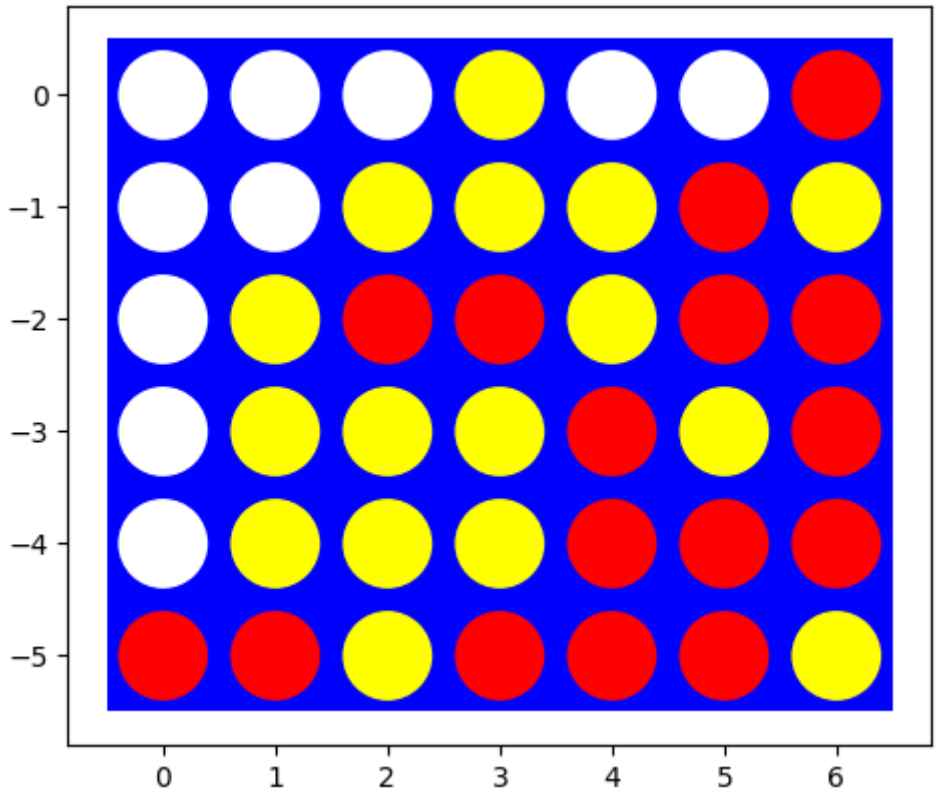

```
Enter the column:6
```



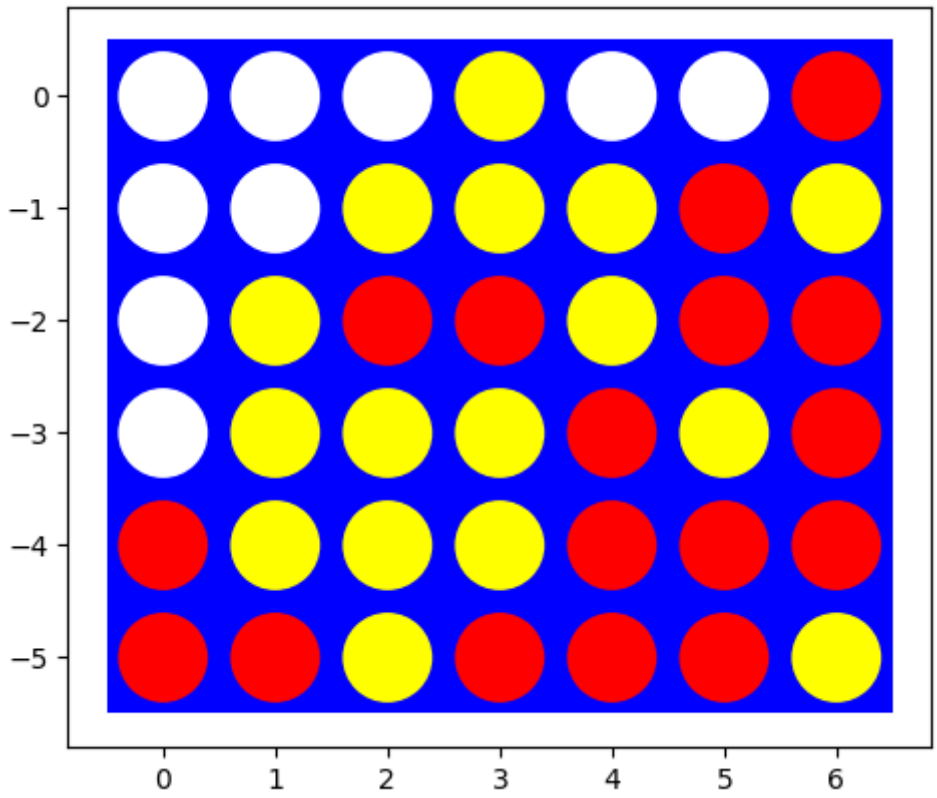



```
Enter the column:0
```

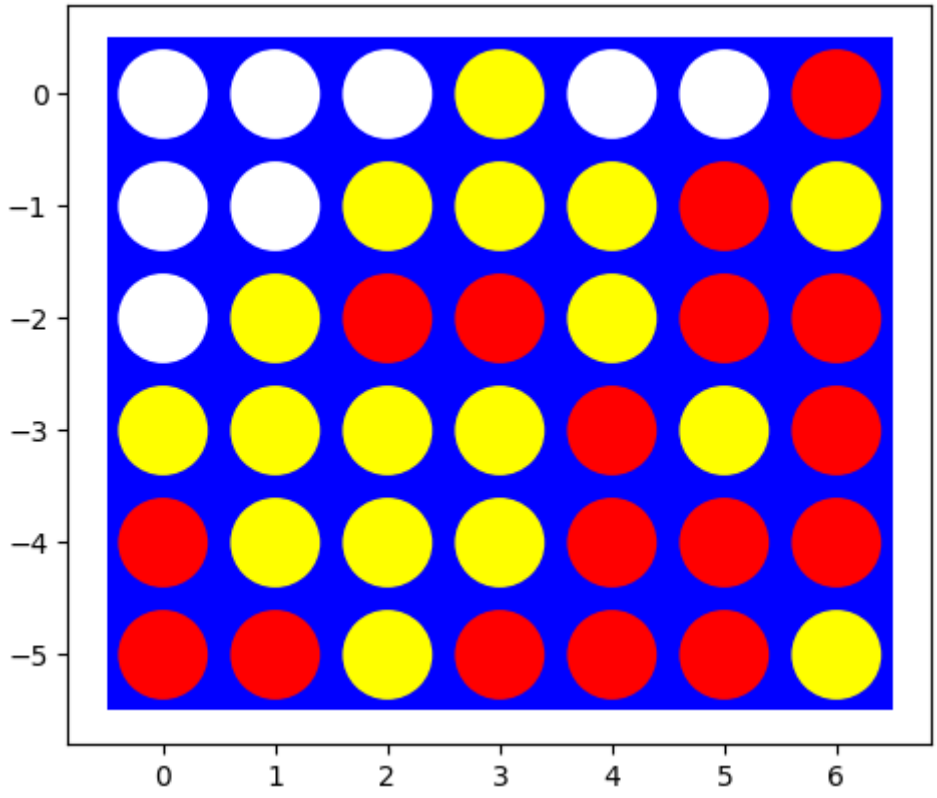





```
Enter the column:0
```




```
AI WIN!
```



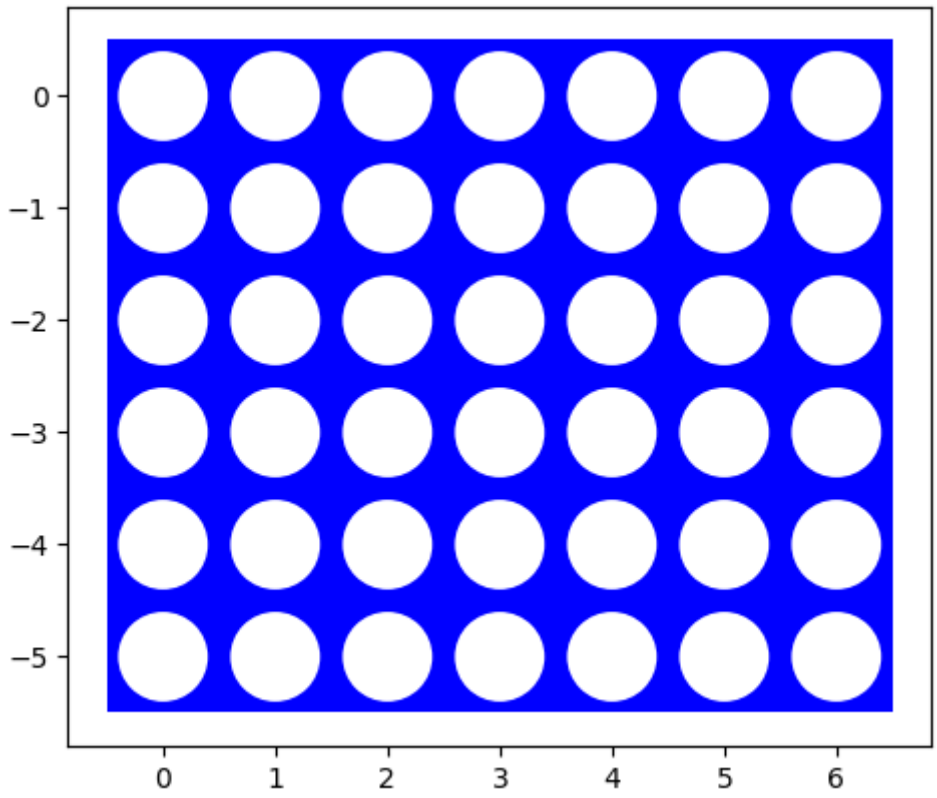
Playtime [5 points]

Let the Minimax Search agent play a random agent on a small board. Analyze wins, losses and draws.

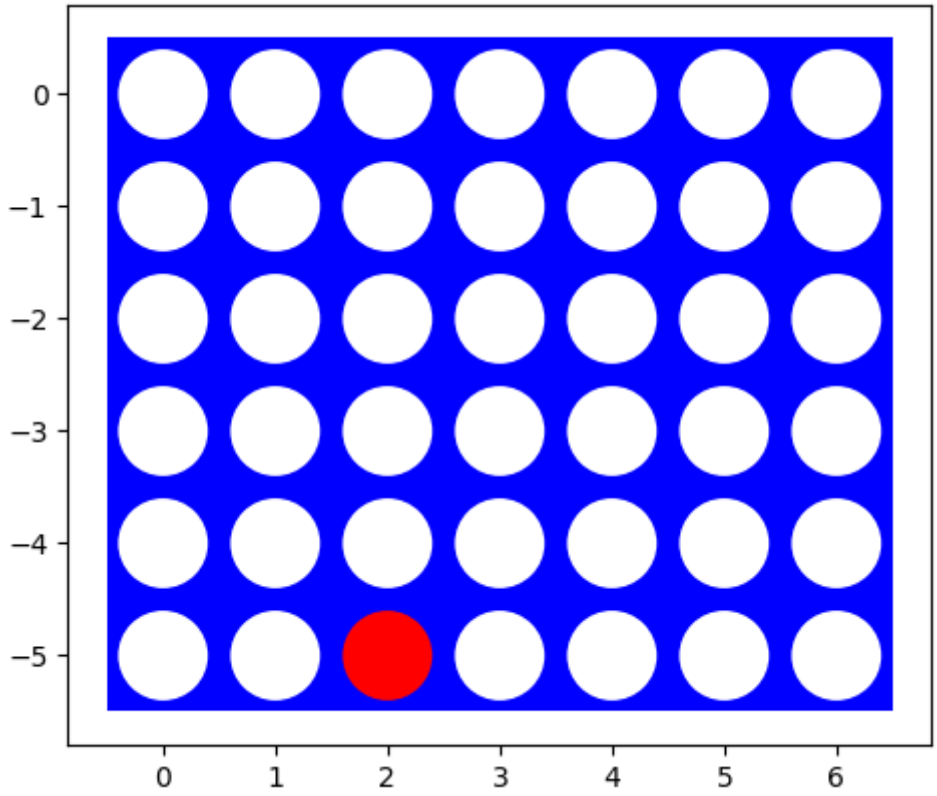
In [23]:

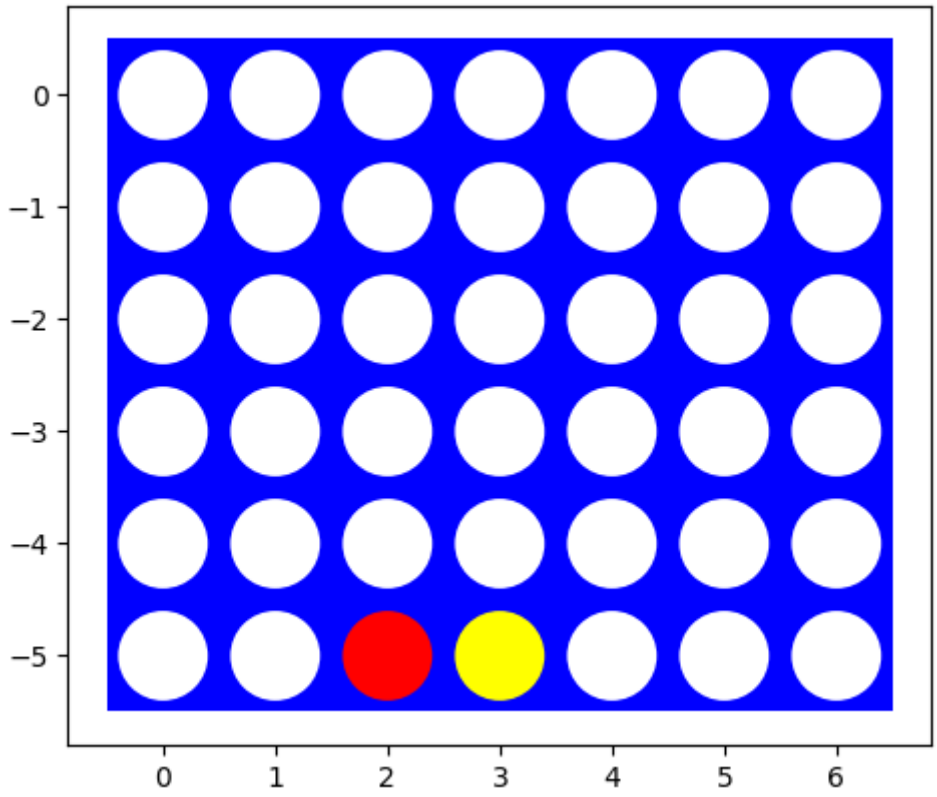
```
game_over_check = False
turn = 0  # Human turn is first
difficulty = 0
```

```
board = initialize_board(6,7)
visualize(board)

while not game_over_check:
    if turn == 0:
        col = int(input("Enter the column:"))
        if valid_actions_check(board, col):
            row = get_row(board, col)
            update_board(board, row, col, 1)
            #print_board(board)
            if check_winner(board,1):
                print("YOU WIN CONGRATS!")
                game_over_check = True
            turn += 1
            visualize(board)
    elif turn == 1:
        func_score, best_col = minmax(board, True, -math.inf, math.inf, 3)
        if valid_actions_check(board, best_col):
            row = get_row(board, best_col)
            update_board(board, row, best_col, 2)
            #print_board(board)
            if check_winner(board, 2):
                print("AI WIN CONGRATS!")
                game_over_check = True
            turn -= 1
            visualize(board)
```

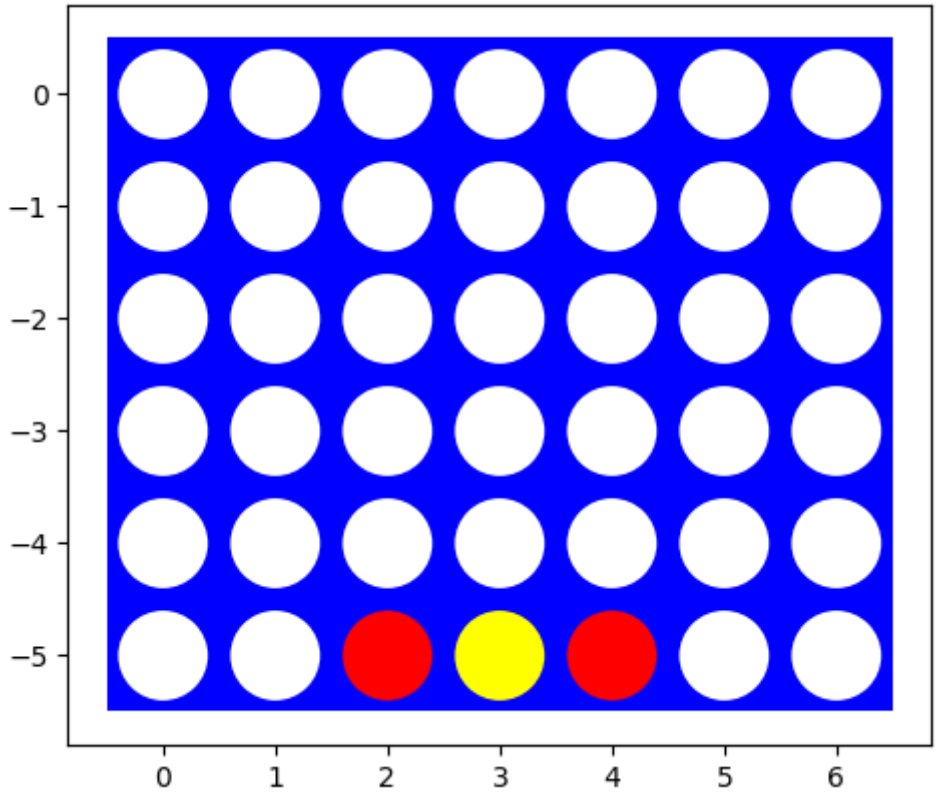


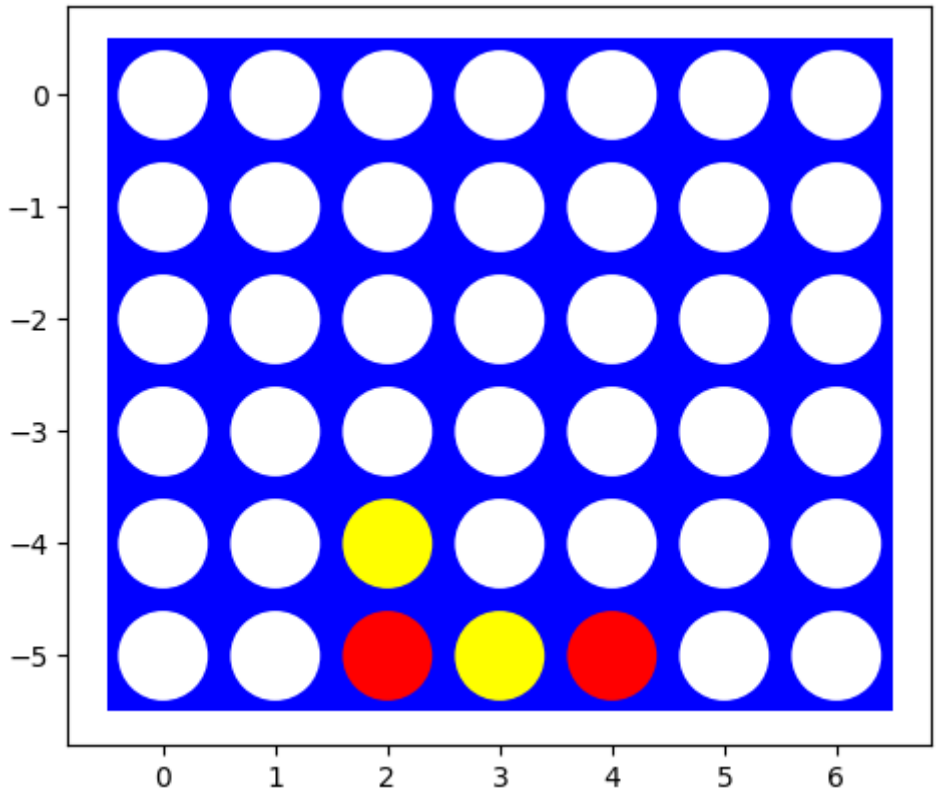

```
Enter the column:2
```



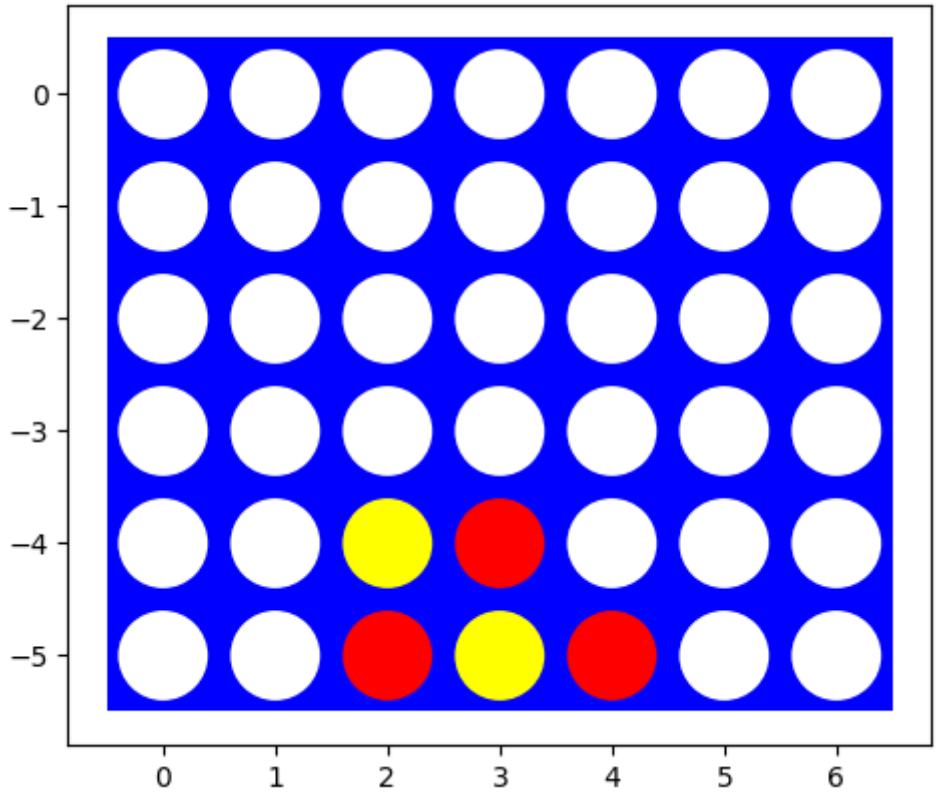


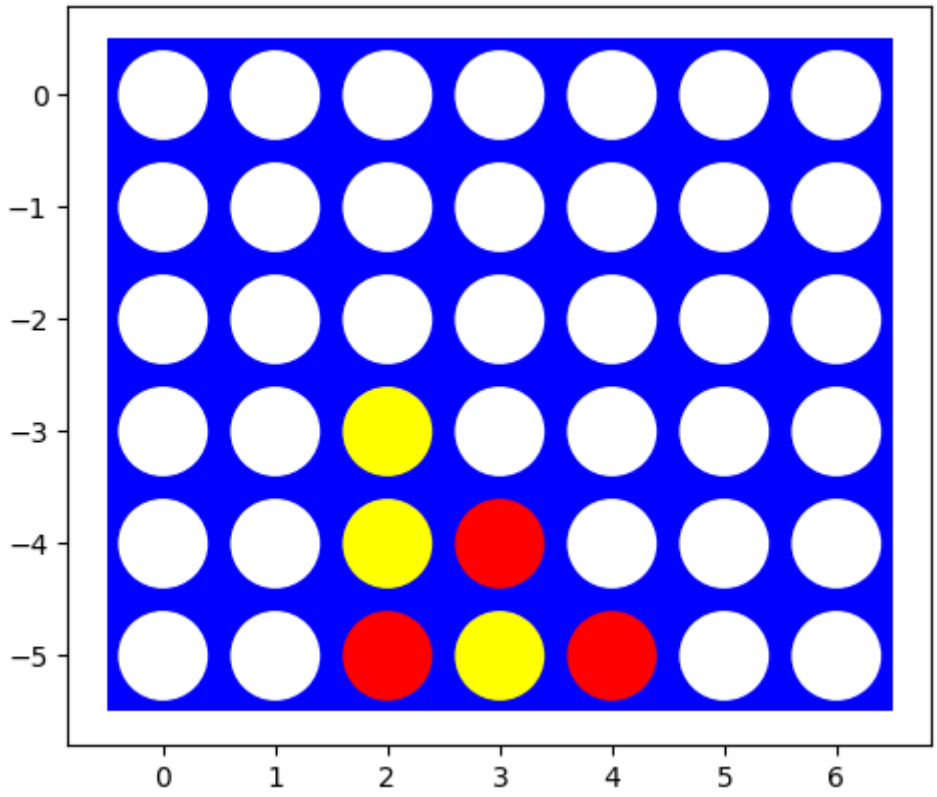

```
Enter the column:4
```



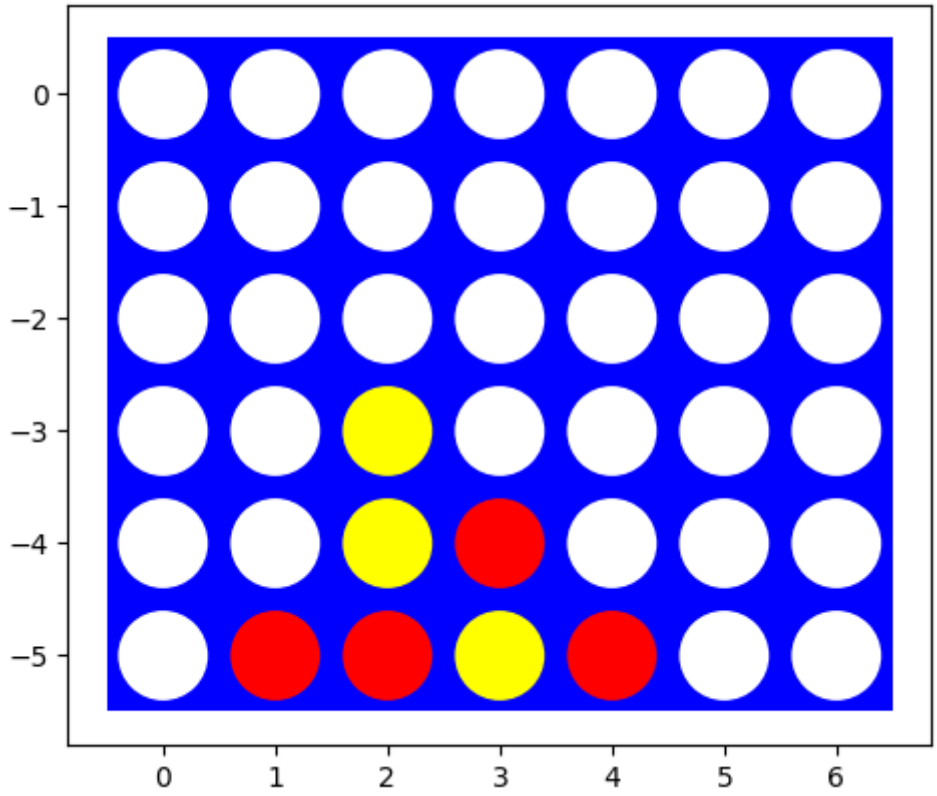


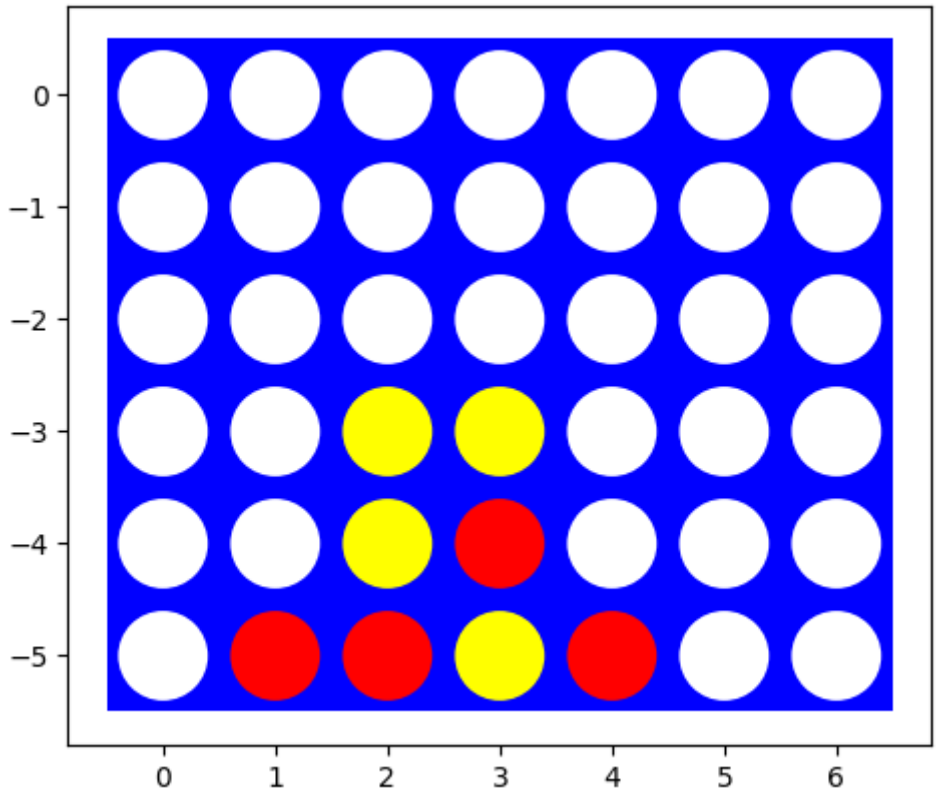

```
Enter the column:3
```



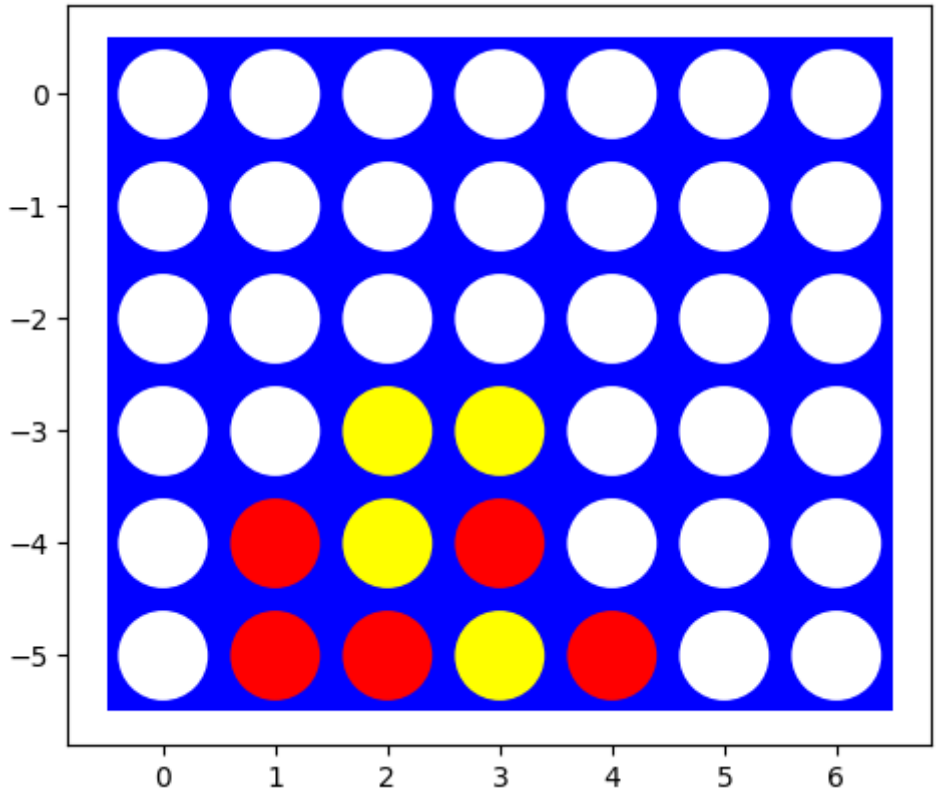


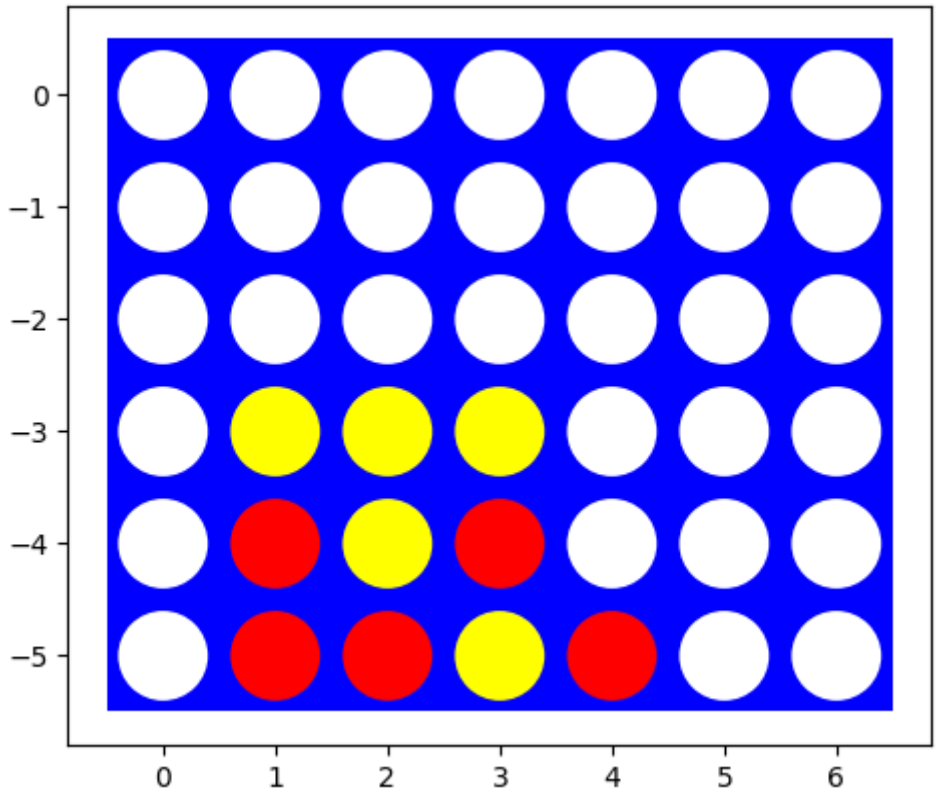

```
Enter the column:1
```



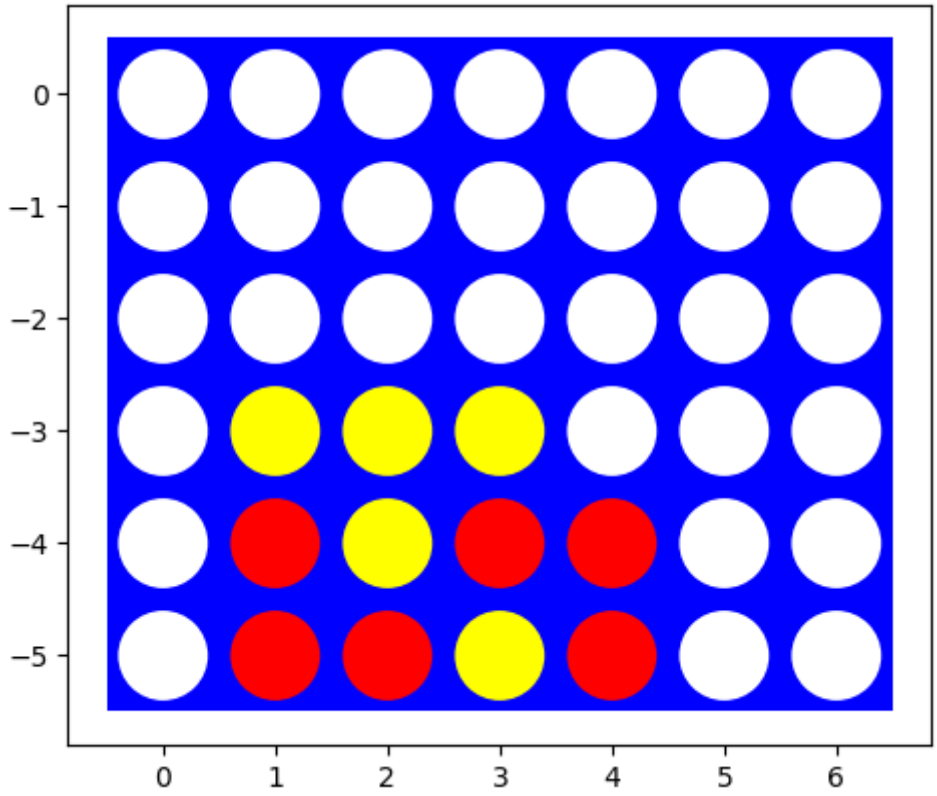


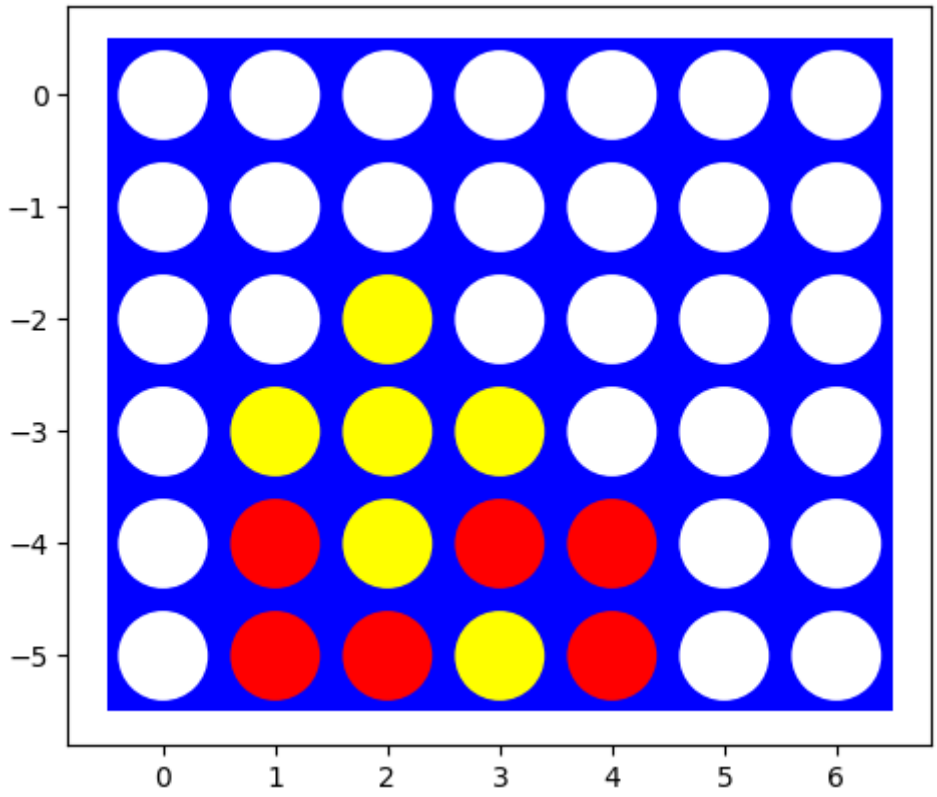

```
Enter the column:1
```



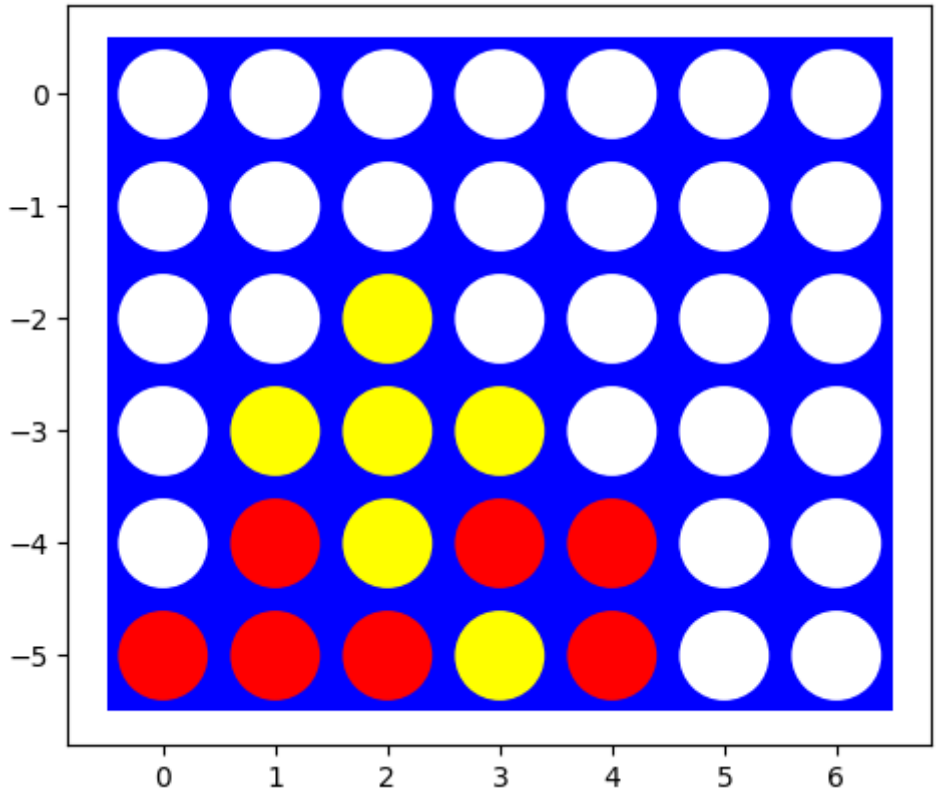


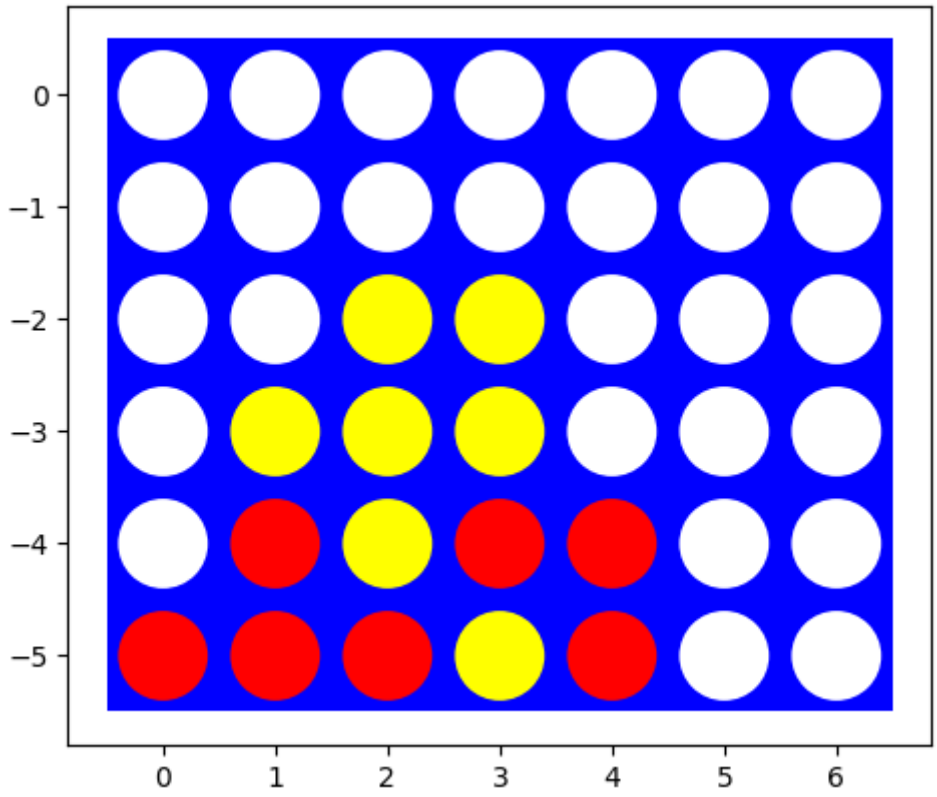

```
Enter the column:4
```



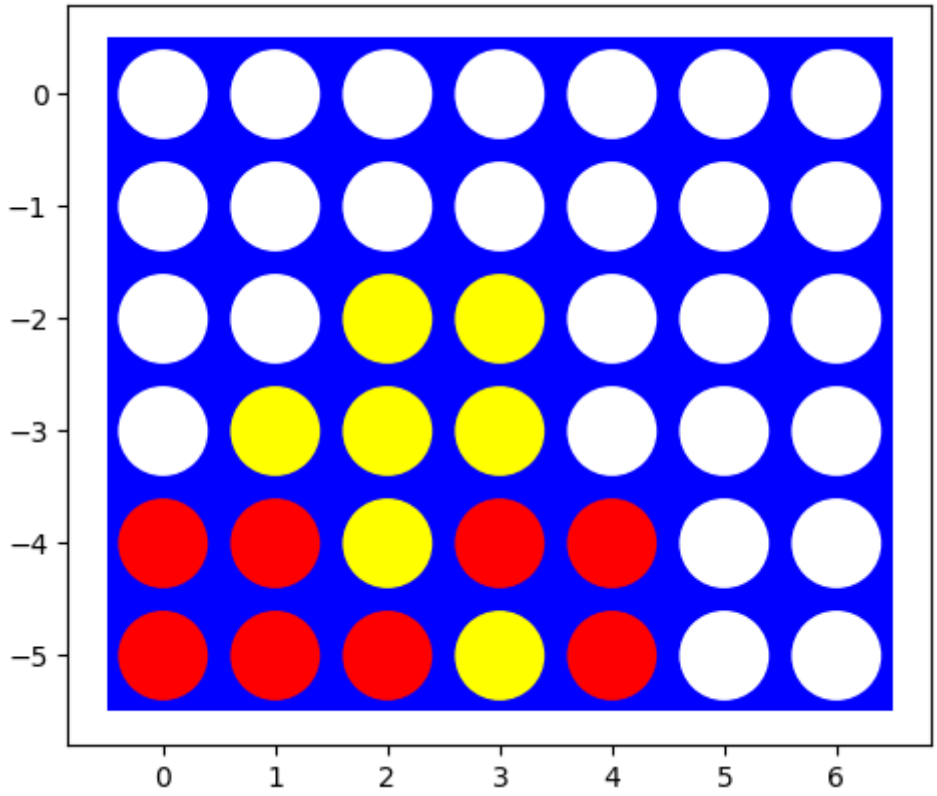


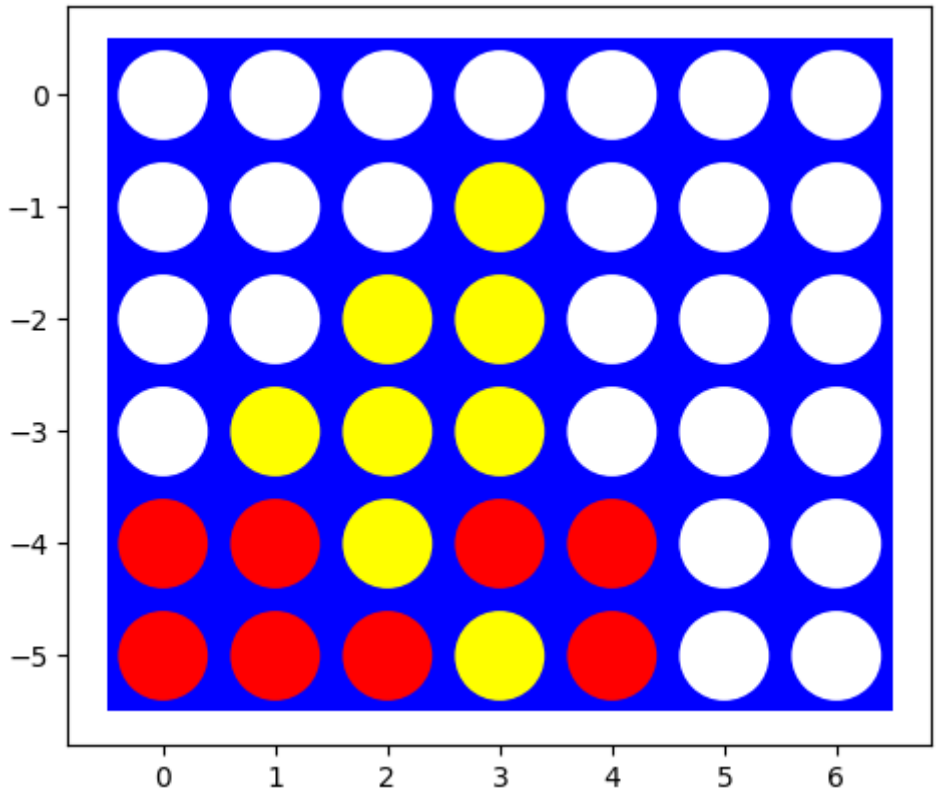

```
Enter the column:0
```



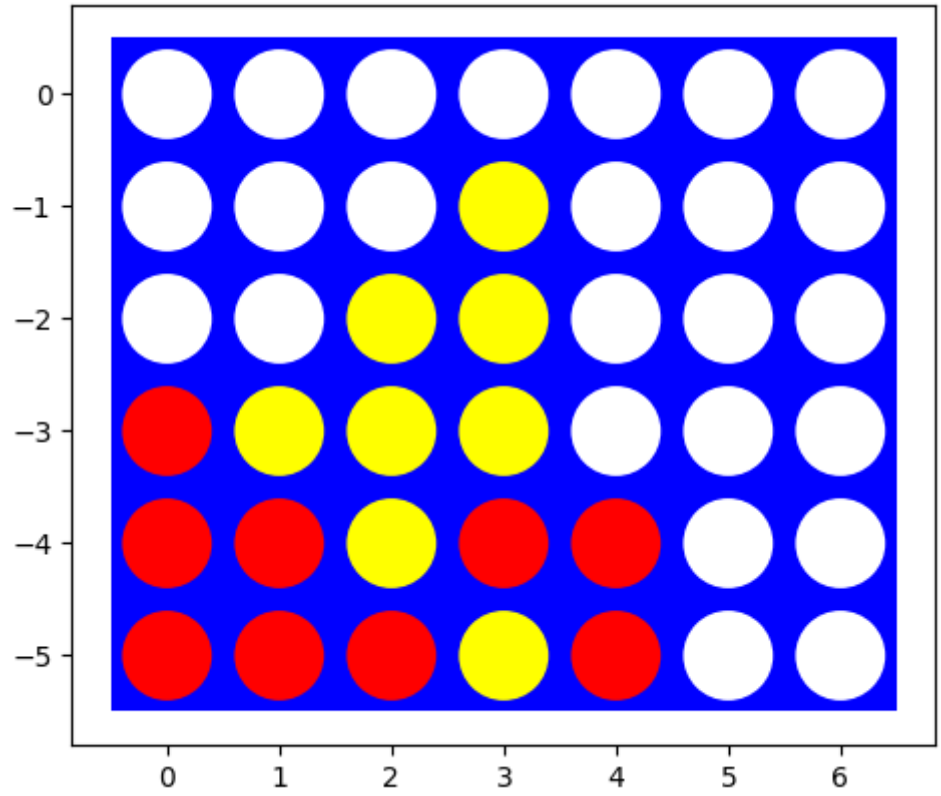



```
Enter the column:0
```

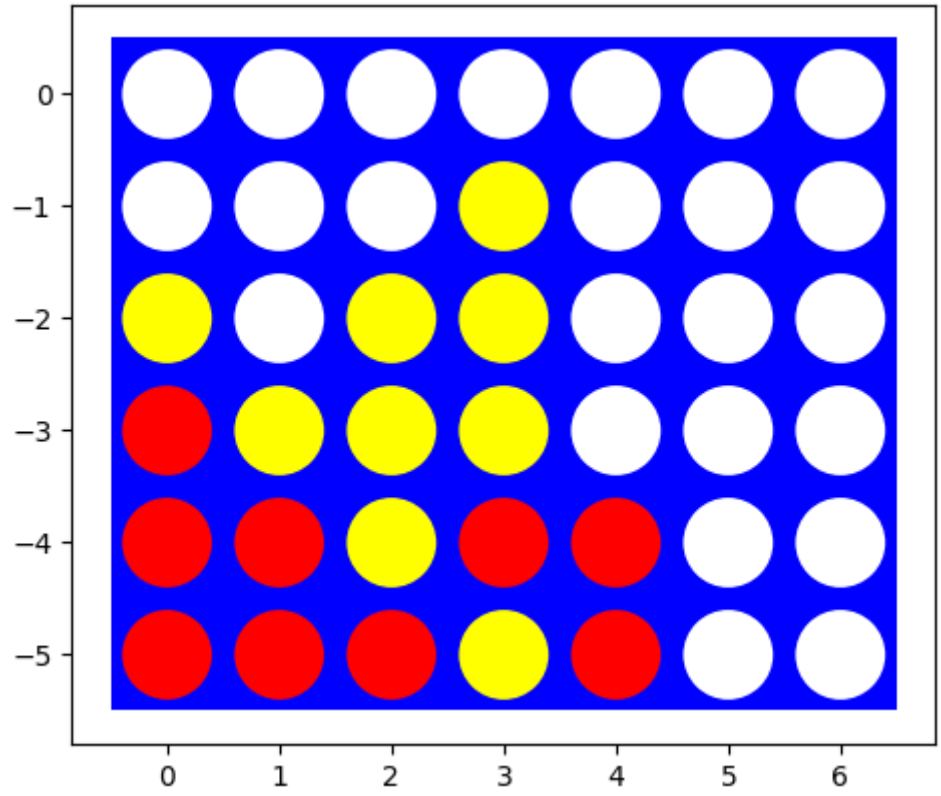





```
Enter the column:0
```




```
AI WIN CONGRATS!
```



Task 4: Heuristic Alpha-Beta Tree Search

Heuristic evaluation function [15 points]

Define and implement a heuristic evaluation function.

heuristic: The heuristic function evaluates the given game board from the perspective of a specific player. It focuses on the center column of the board and counts the number of pieces

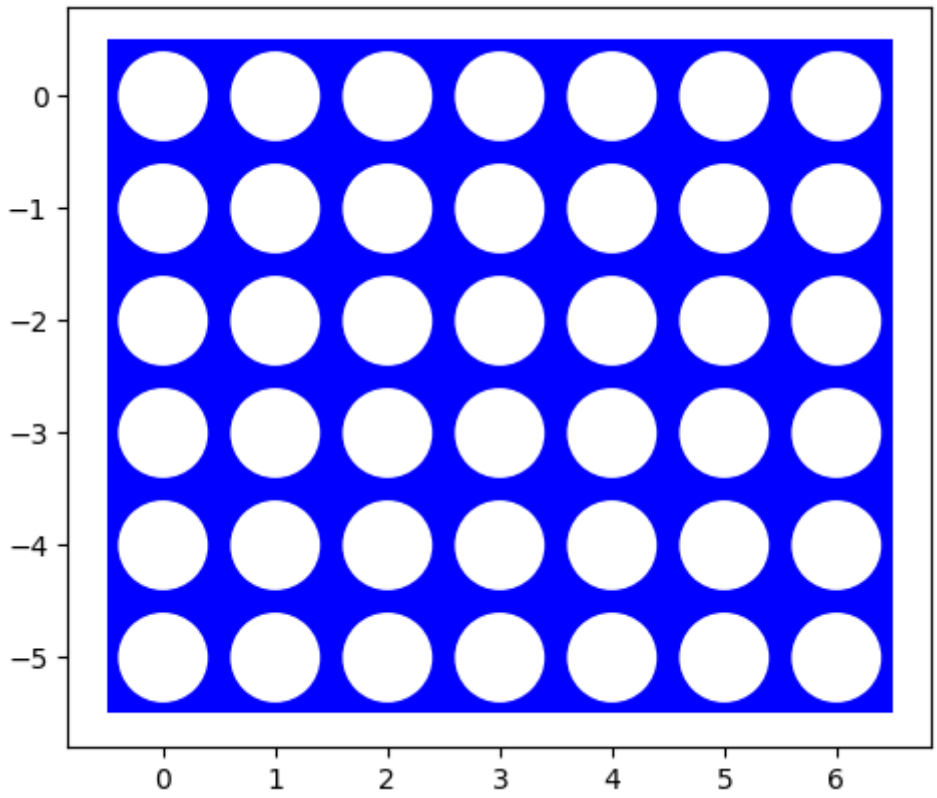
belonging to the specified player in that column.

In [26]:

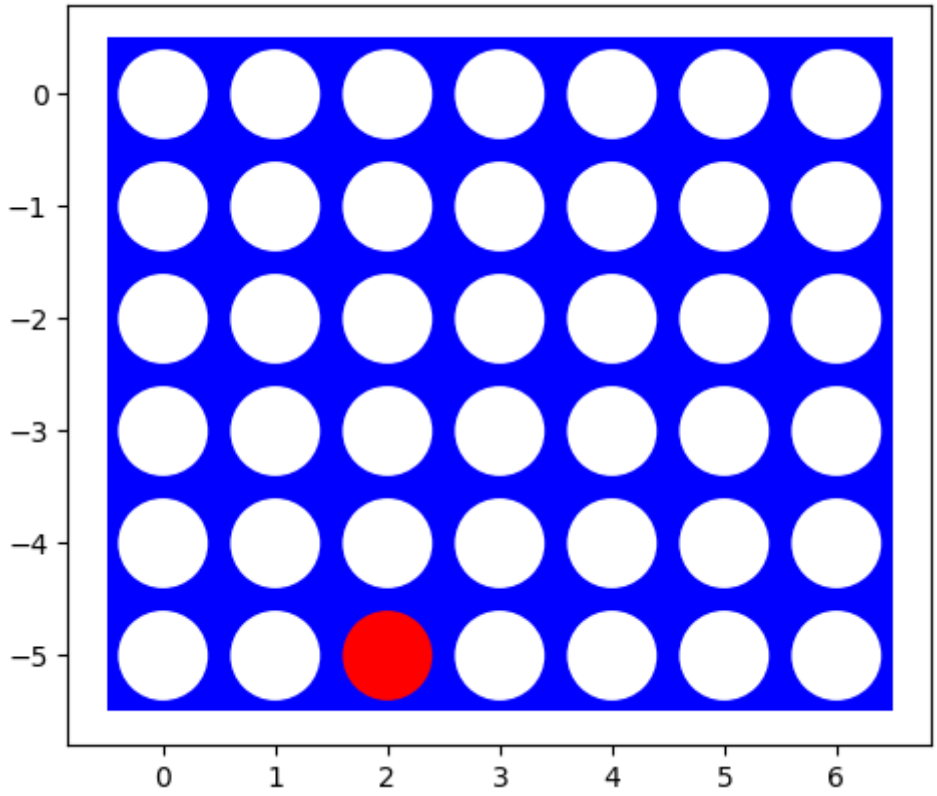
```
def heuristic(board, player):
    center_column = board[:, len(board[0]) // 2]
    player_pieces_in_center = center_column.tolist().count(player)
    return player_pieces_in_center

def heuristic_alpha_beta_search(board, player, depth, alpha, beta):
    def max_value(board, player, depth, alpha, beta):
        if depth == 0 or is_terminal_state(board):
            return heuristic(board, player)

        value = -math.inf
        valid_moves = valid_actions(board)

        for move in valid_moves:
            new_board = copy.deepcopy(board)
            make_move(new_board, move, player)
            value = max(value, min_value(new_board, player, depth - 1, alpha, beta))

            alpha = max(alpha, value)
            if alpha >= beta:
                break

        return value

    def min_value(board, player, depth, alpha, beta):
        if depth == 0 or is_terminal_state(board):
            return heuristic(board, player)

        value = math.inf
        valid_moves = valid_actions(board)

        for move in valid_moves:
            new_board = copy.deepcopy(board)
            make_move(new_board, move, player)
            value = min(value, max_value(new_board, player, depth - 1, alpha, beta))

            beta = min(beta, value)
            if alpha >= beta:
                break

        return value

    best_move = None
    max_score = -math.inf
    valid_moves = valid_actions(board)

    for move in valid_moves:
        new_board = copy.deepcopy(board)
        make_move(new_board, move, player)
        score = min_value(new_board, player, depth - 1, alpha, beta)

        if score > max_score:
            max_score = score
            best_move = move

        alpha = max(alpha, max_score)
        if alpha >= beta:
            break

    return best_move
```

In [ ]:

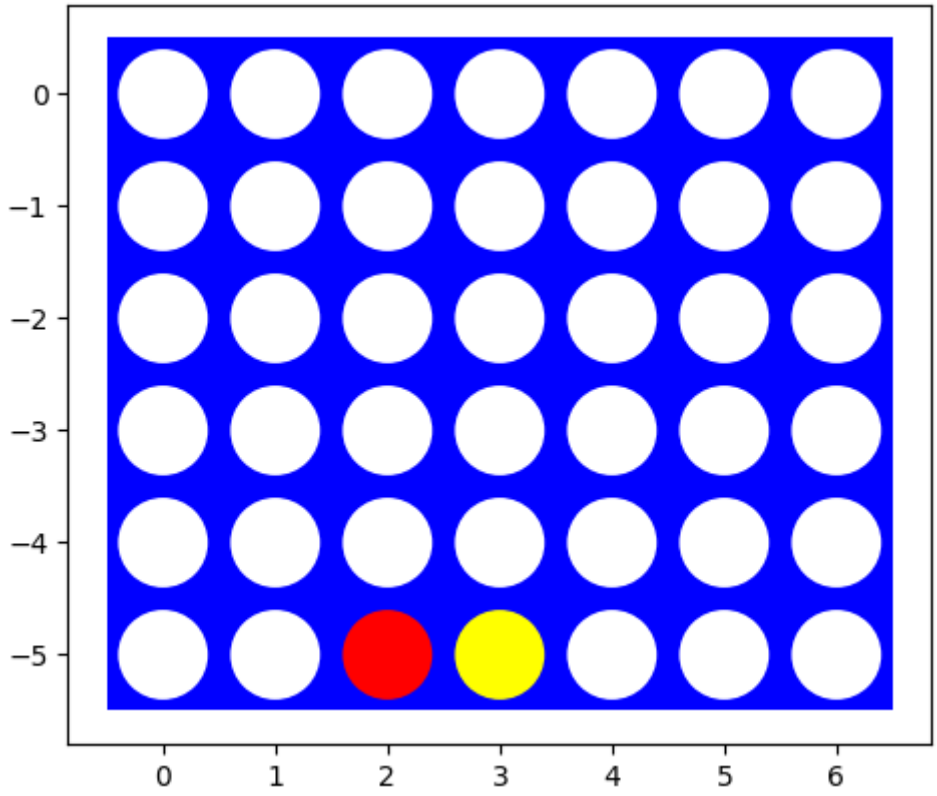
```
board = initialize_board(rows, cols)

while not is_terminal_state(board):
  visualize(board)
  action = int(input("Enter your column: "))
  if valid_actions_check(board, action):
    make_move(board, action, 1)
  if is_terminal_state(board): break
  visualize(board)

  print("AI's move:")
  ai_move = heuristic_alpha_beta_search(board, 2, depth=3, alpha=-math.inf, beta=mat
  make_move(board, ai_move, 2)
visualize(board)
```



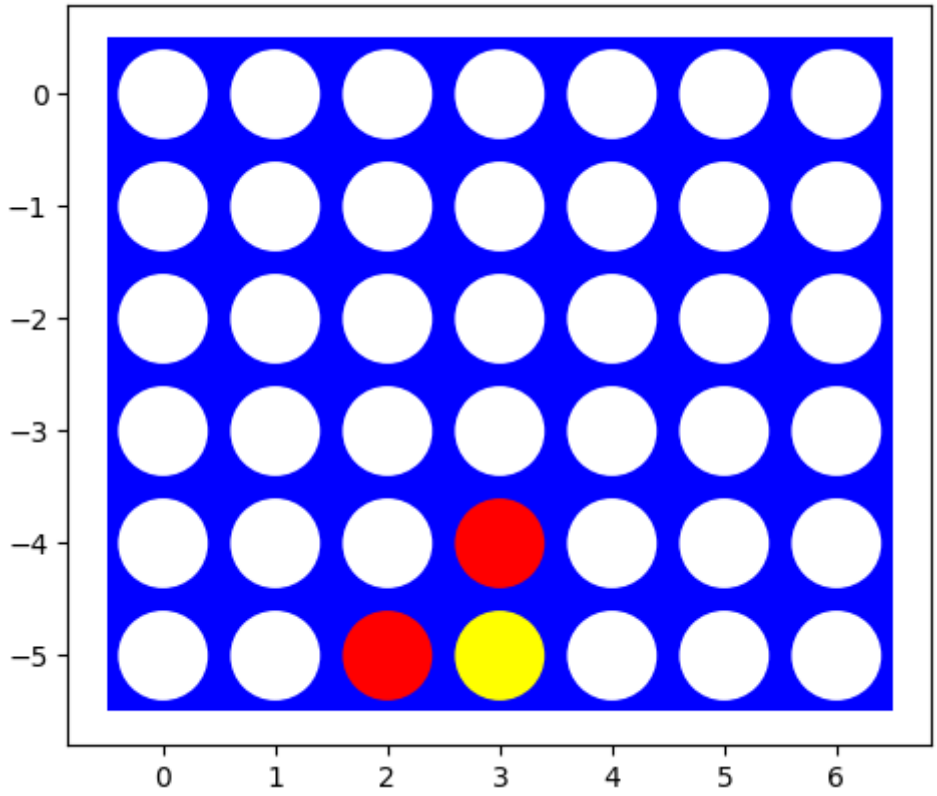

```
Enter your column: 2
```



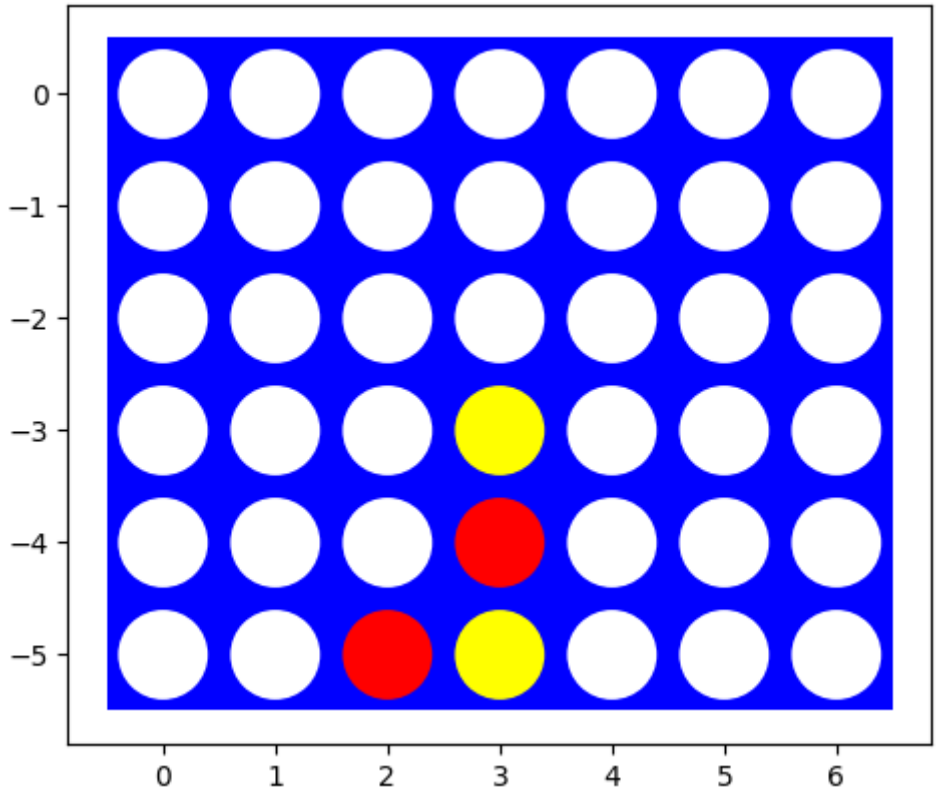
AI's move:



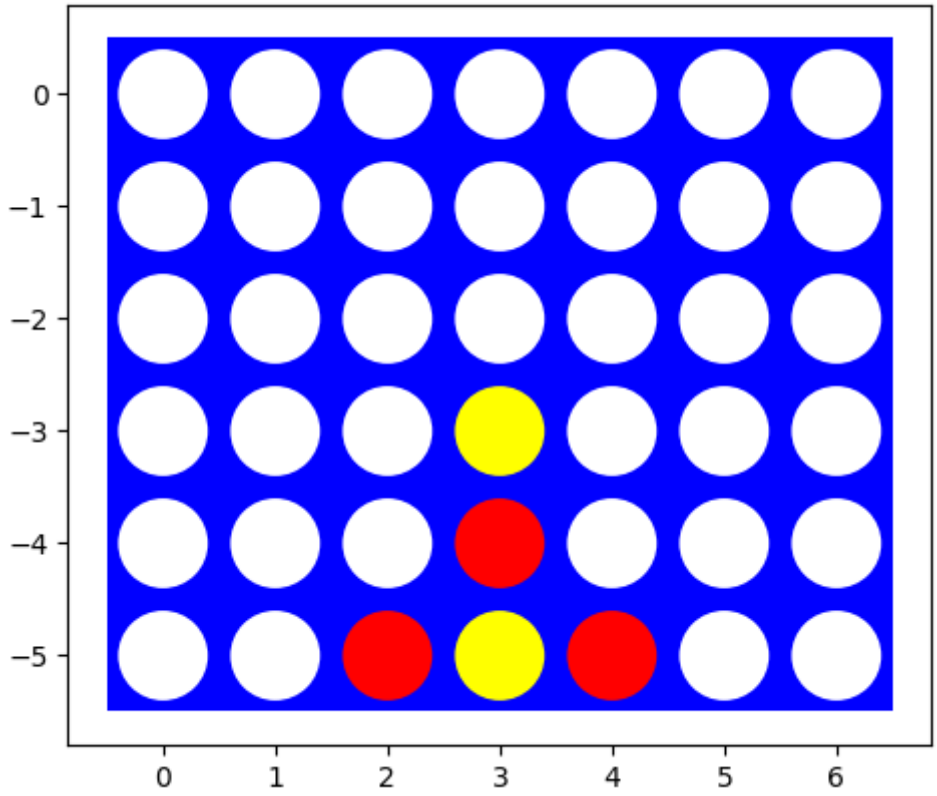
Enter your column: 3



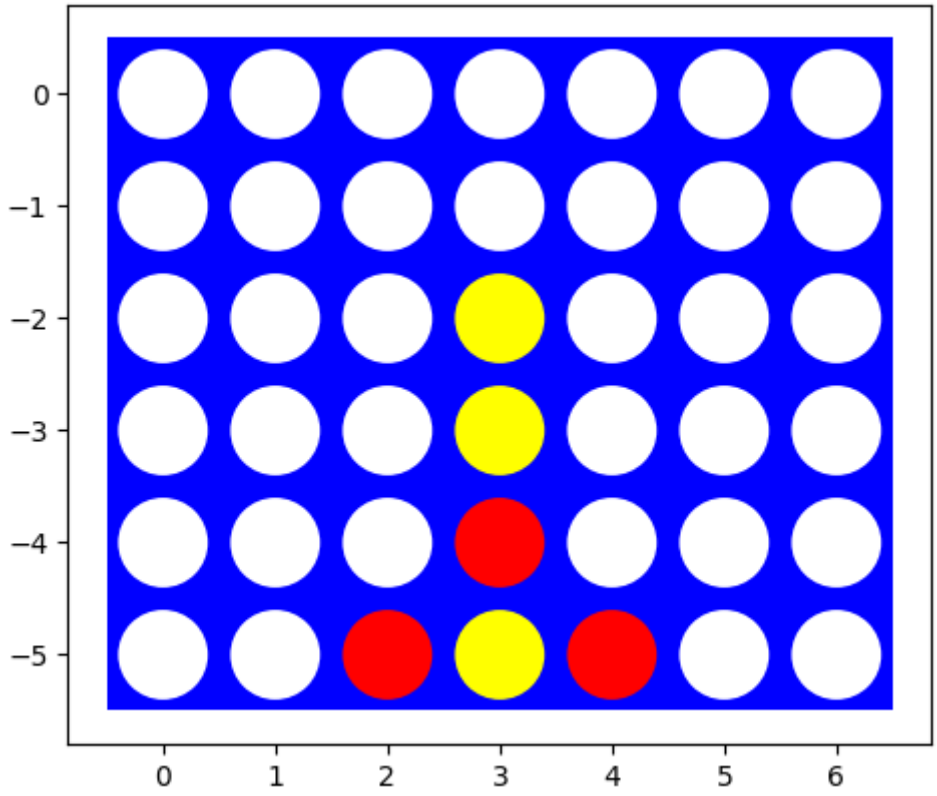
AI's move:



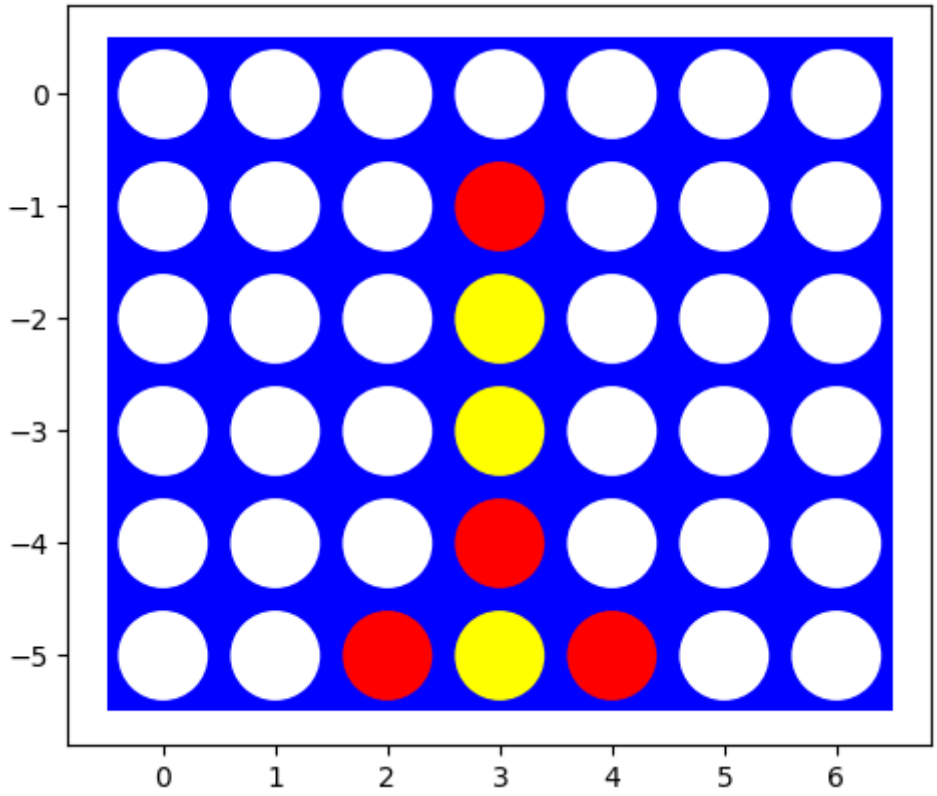
Enter your column: 4



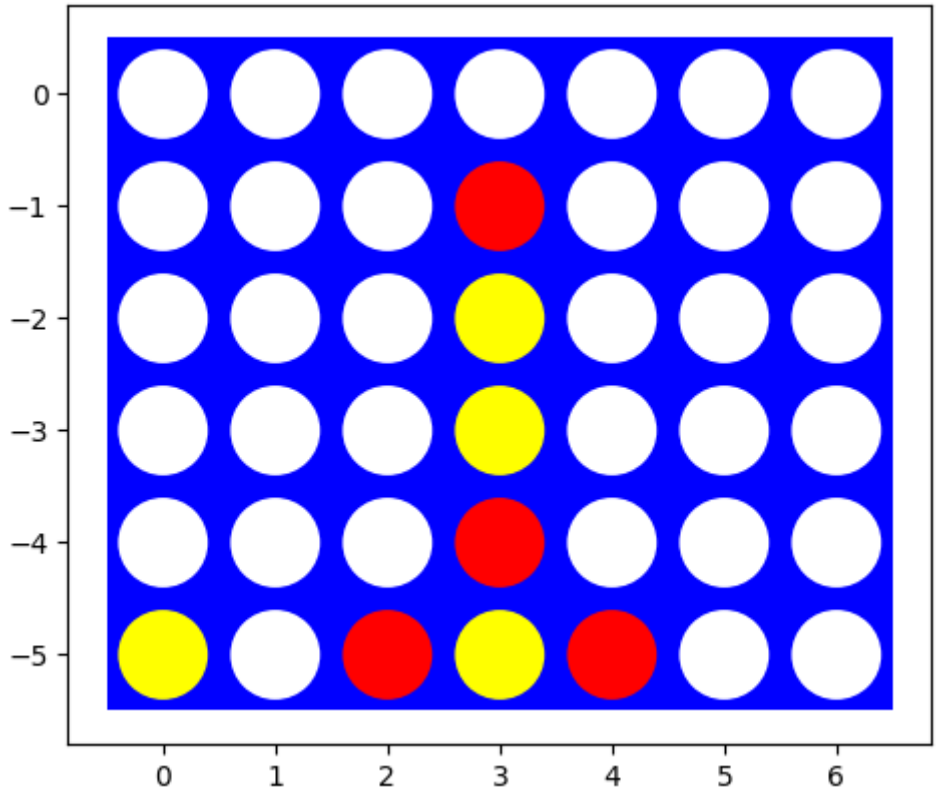
AI's move:



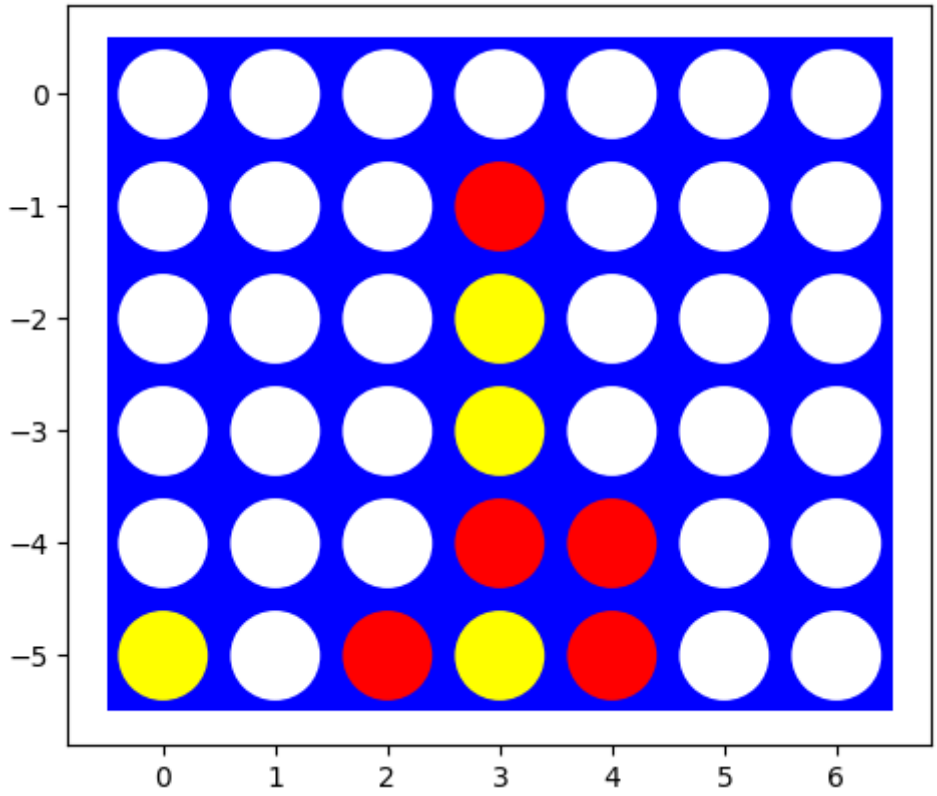
Enter your column: 3



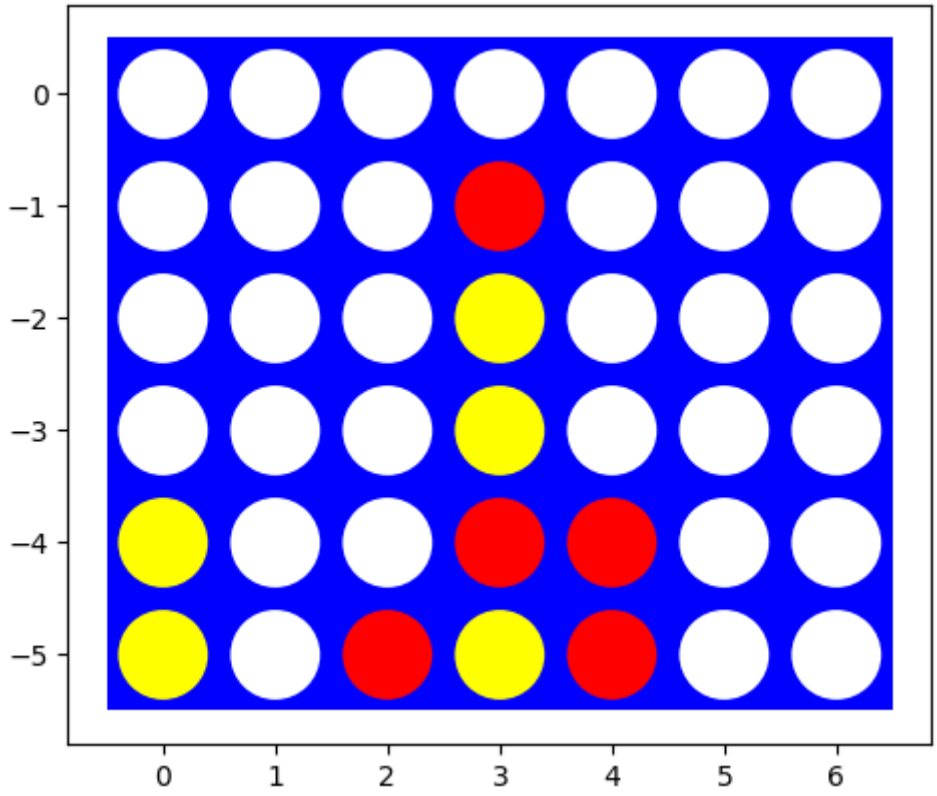

```
AI's move:
```



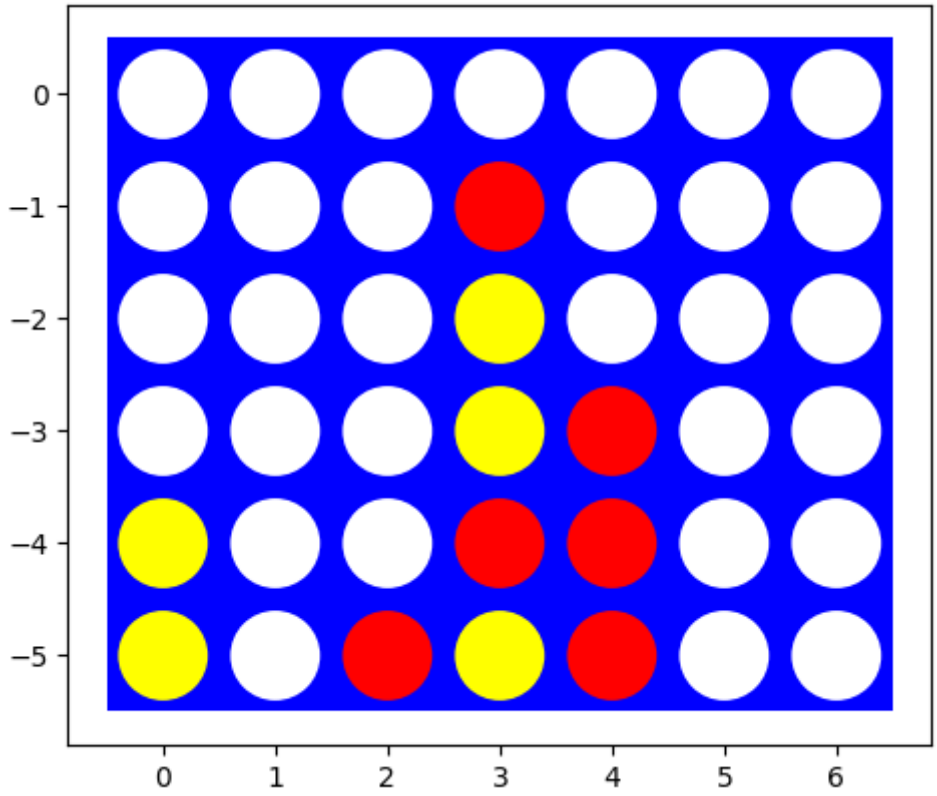

```
Enter your column: 4
```



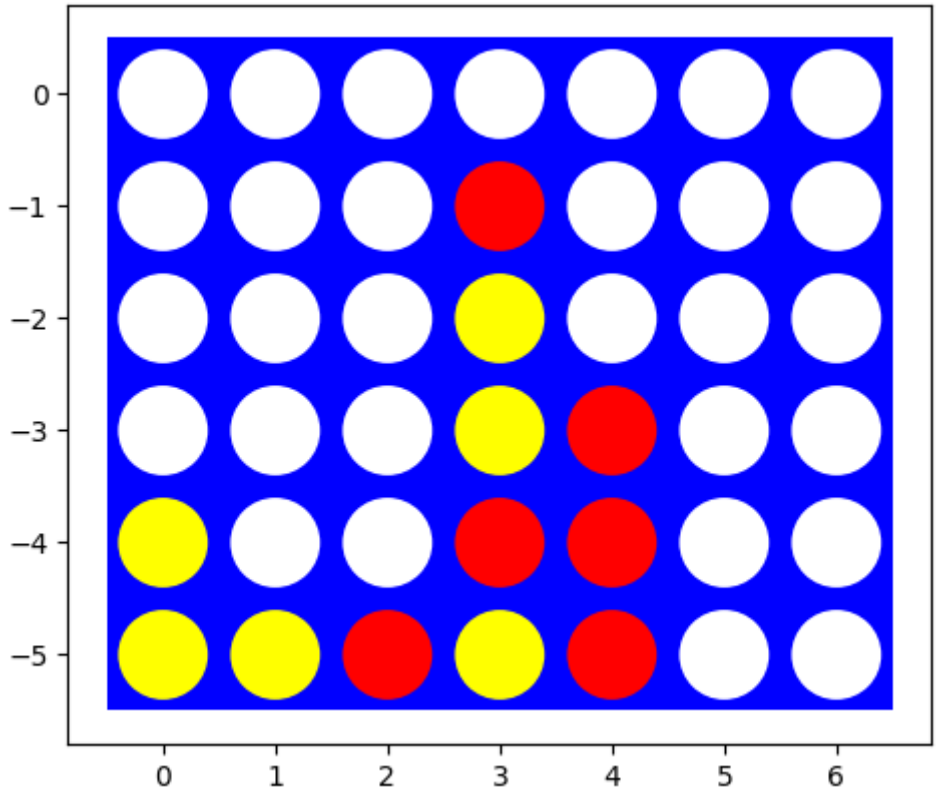

```
AI's move:
```



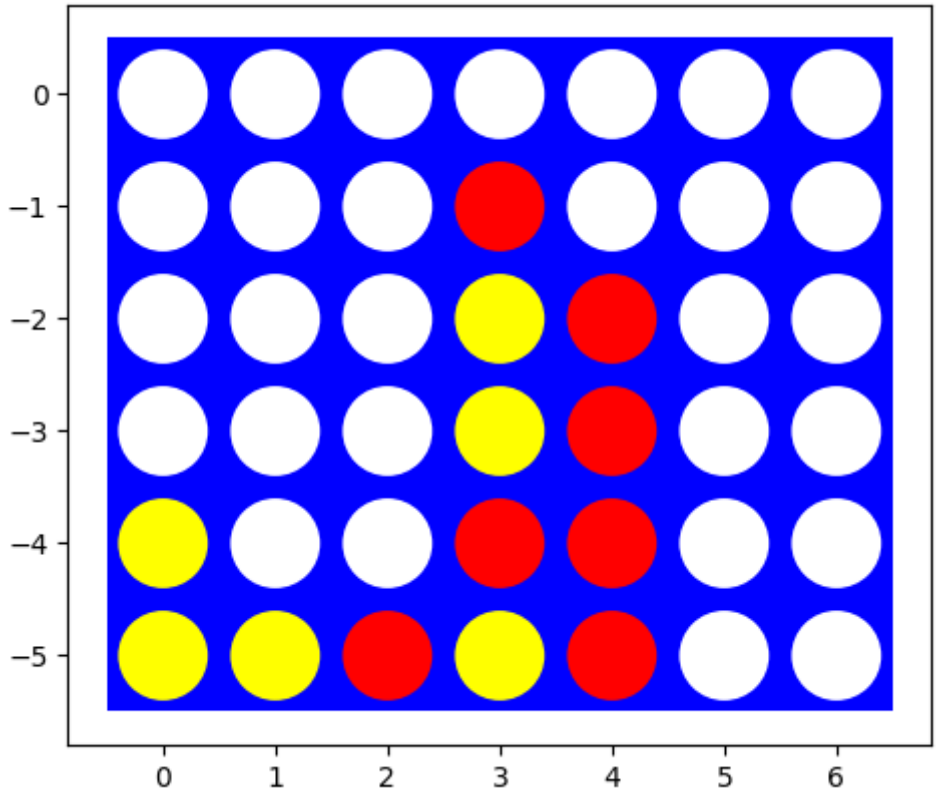

```
Enter your column: 4
```



AI's move:



Enter your column: 4



Cutting off search [10 points]

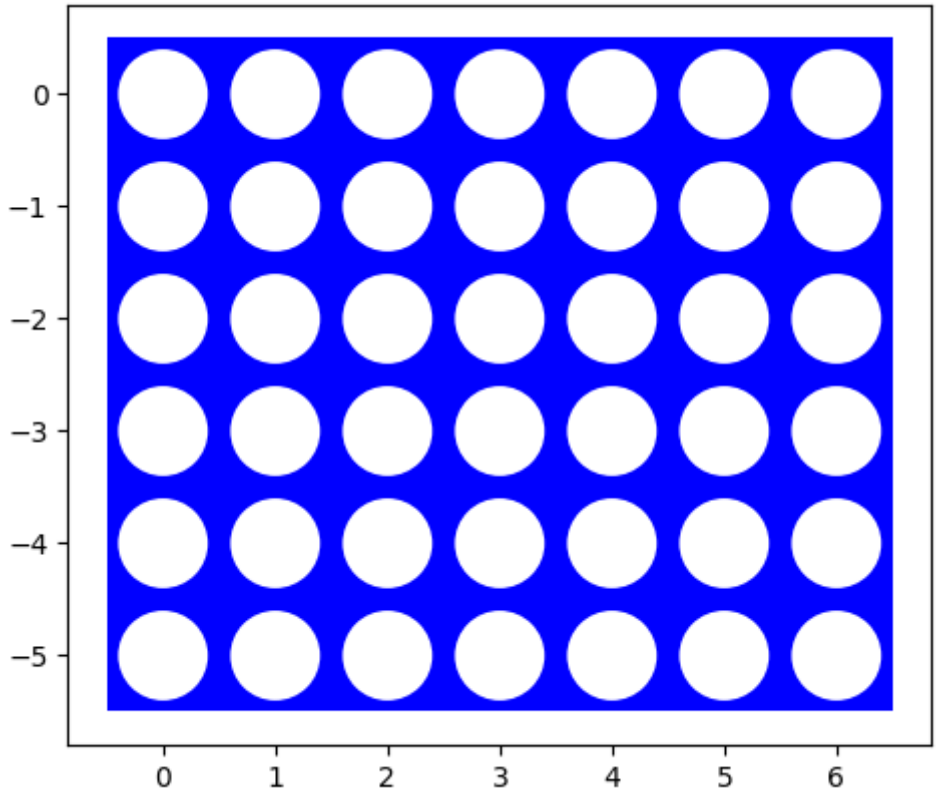
Modify your Minimax Search with Alpha-Beta Pruning to cut off search at a specified depth and use the heuristic evaluation function. Experiment with different cutoff values.

heuristic_alpha_beta_search: The function performs an alpha-beta search to determine the best move for the AI player given the current state of the game. It uses nested helper functions for maximum (max_value) and minimum (min_value) values in the alpha-beta search. The function returns the best move determined by the alpha-beta search after evaluating possible moves based on the heuristic.

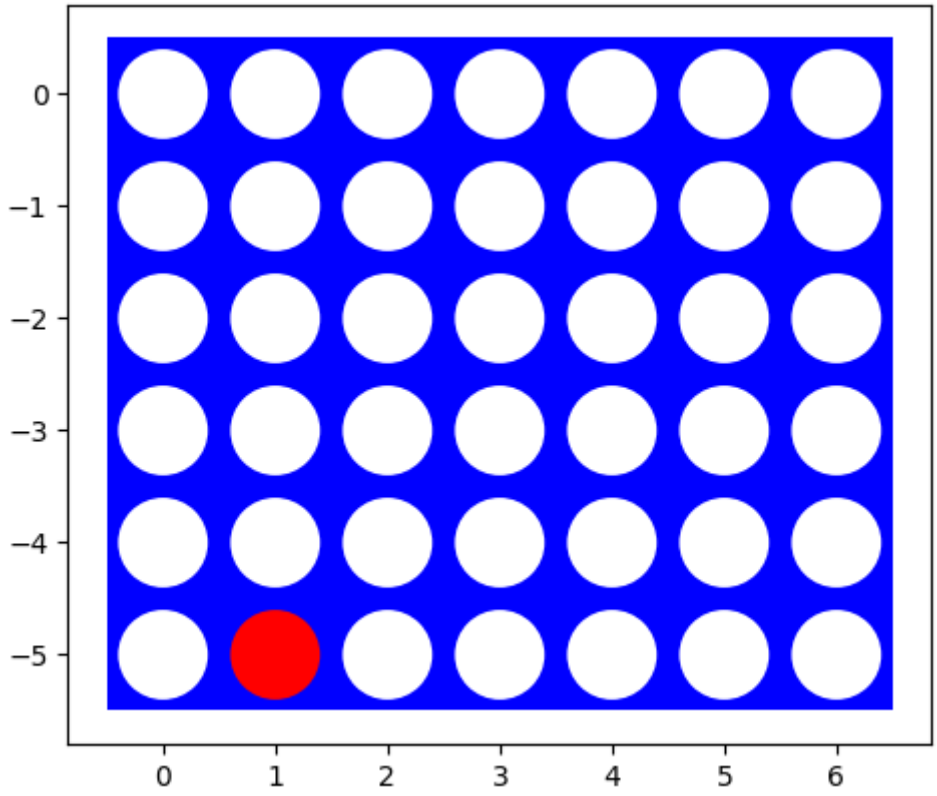
In [27]:

```
import copy
import math

rows, cols = 6, 7

def heuristic_alpha_beta_search(board, player, depth, alpha, beta):
    def max_value(board, player, depth, alpha, beta):
        if depth == 0 or is_terminal_state(board):
            return heuristic(board, player)

        value = -math.inf
        valid_moves = valid_actions(board)

        for move in valid_moves:
            new_board = copy.deepcopy(board)
            make_move(new_board, move, player)
            value = max(value, min_value(new_board, player, depth - 1, alpha, beta))

            alpha = max(alpha, value)
            if alpha >= beta:
                break

        return value
```

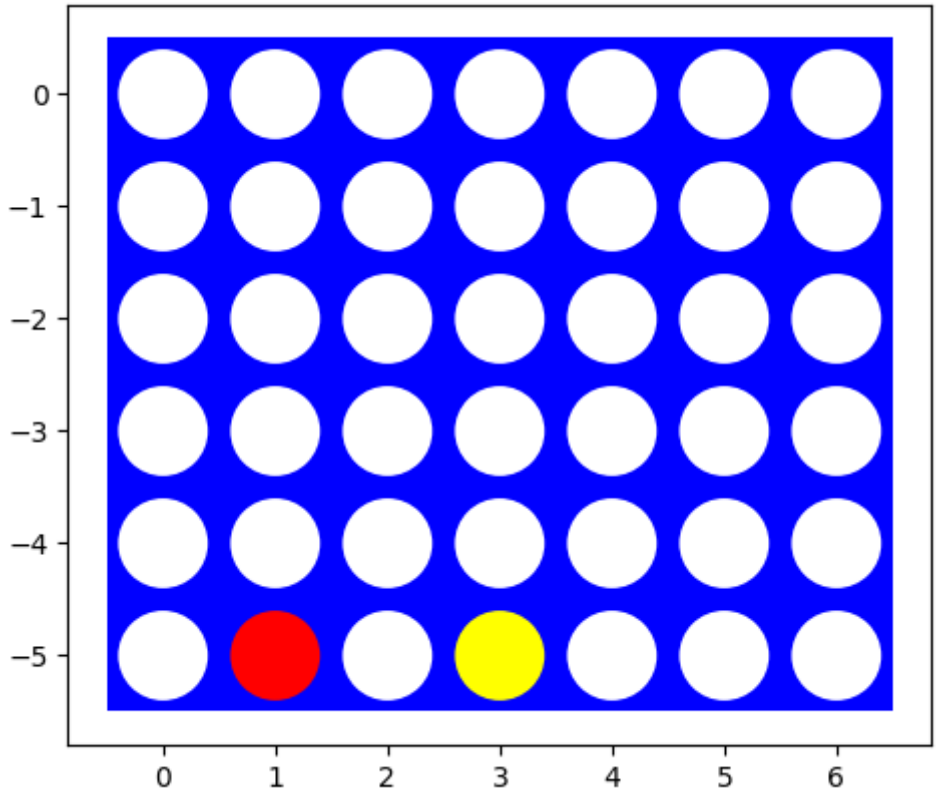
```
    def min_value(board, player, depth, alpha, beta):
        if depth == 0 or is_terminal_state(board):
            return heuristic(board, player)

        value = math.inf
        valid_moves = valid_actions(board)

        for move in valid_moves:
            new_board = copy.deepcopy(board)
            make_move(new_board, move, player)
            value = min(value, max_value(new_board, player, depth - 1, alpha, beta))

            beta = min(beta, value)
            if alpha >= beta:
                break

        return value

    best_move = None
    max_score = -math.inf
    valid_moves = valid_actions(board)

    for move in valid_moves:
        new_board = copy.deepcopy(board)
        make_move(new_board, move, player)
        score = min_value(new_board, player, depth - 1, alpha, beta)

        if score > max_score:
            max_score = score
            best_move = move

        alpha = max(alpha, max_score)
        if alpha >= beta:
            break

    return best_move

board = initialize_board(rows, cols)

while not is_terminal_state(board):
    visualize(board)
    action = int(input("Enter your column: "))
    if valid_actions_check(board, action):
        make_move(board, action, 1)
    if is_terminal_state(board):
        break
    visualize(board)

    print("AI's move:")
    # Experiment with different cutoff values by adjusting the depth parameter
    ai_move = heuristic_alpha_beta_search(board, 2, depth=3, alpha=-math.inf, beta=m
    make_move(board, ai_move, 2)

visualize(board)
```



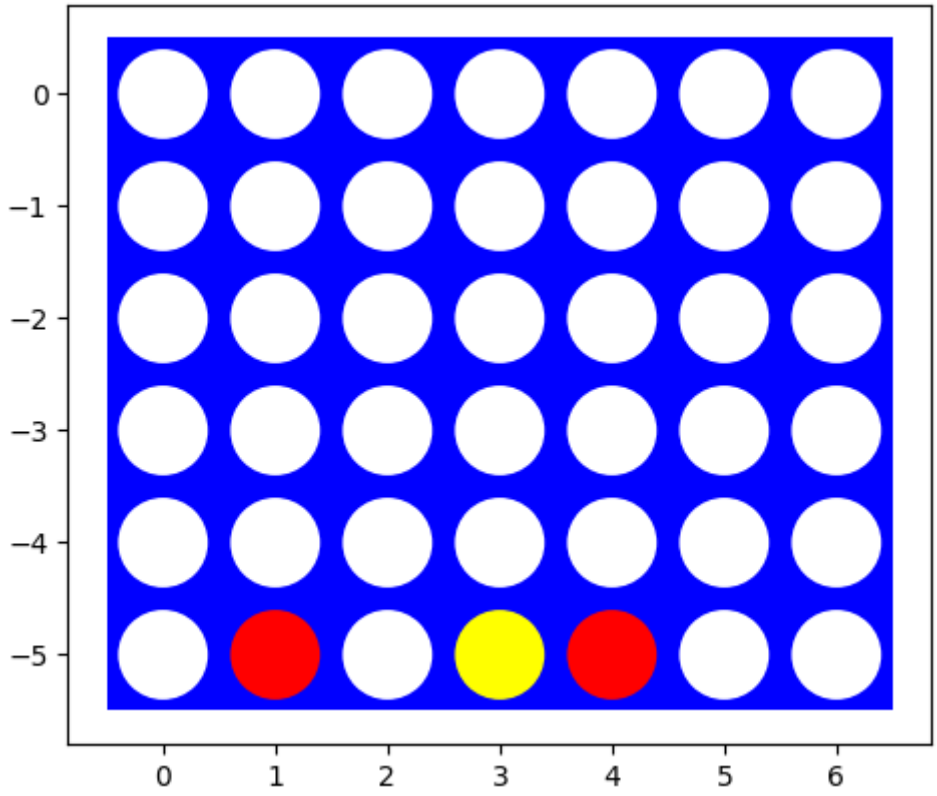

```
Enter your column: 1
```



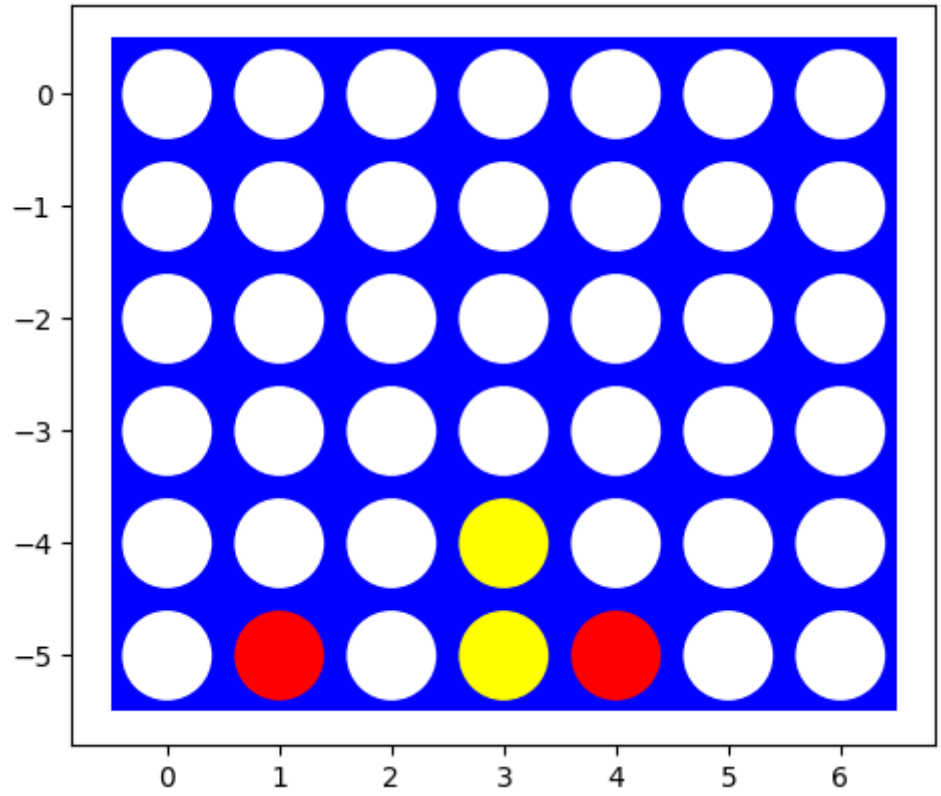

```
AI's move:
```



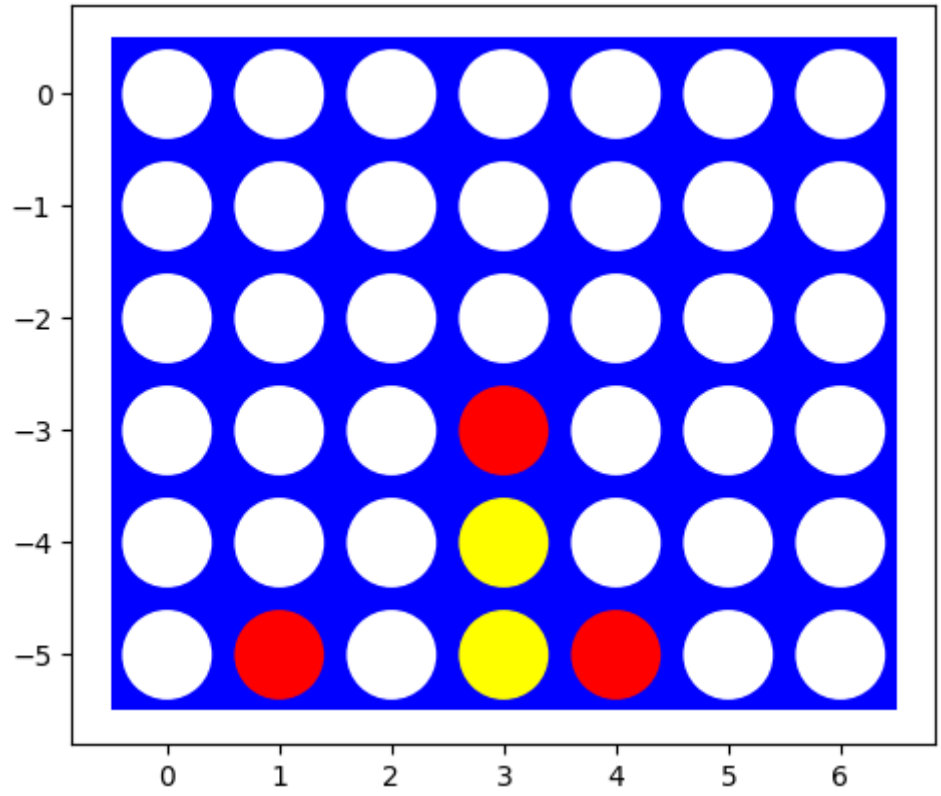

```
Enter your column: 4
```



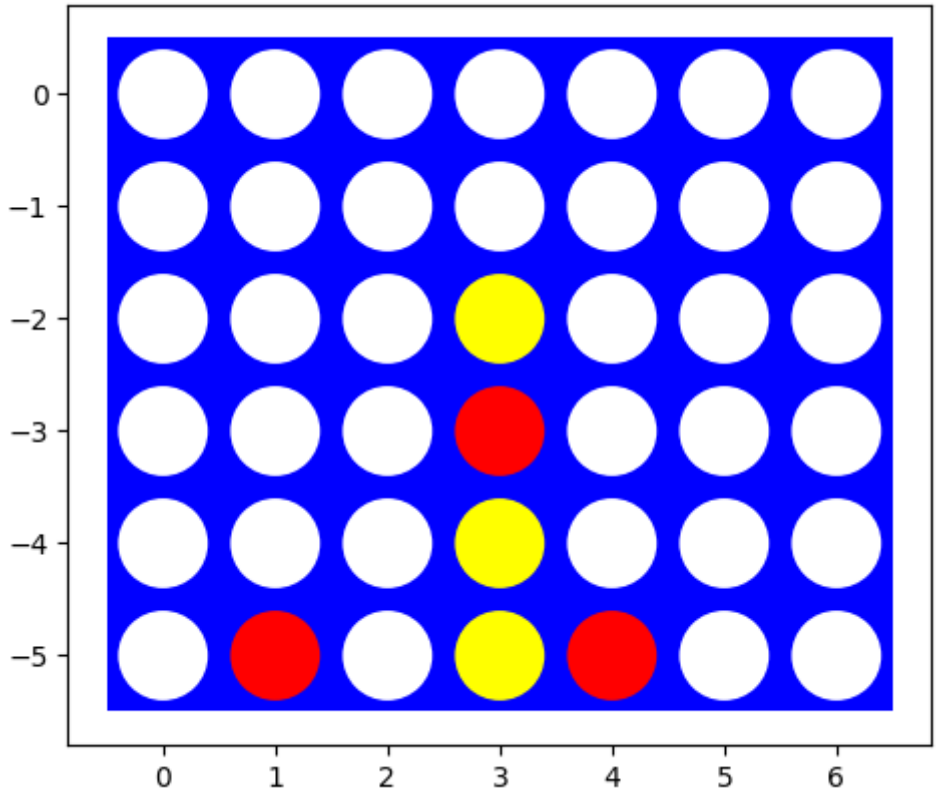

```
AI's move:
```



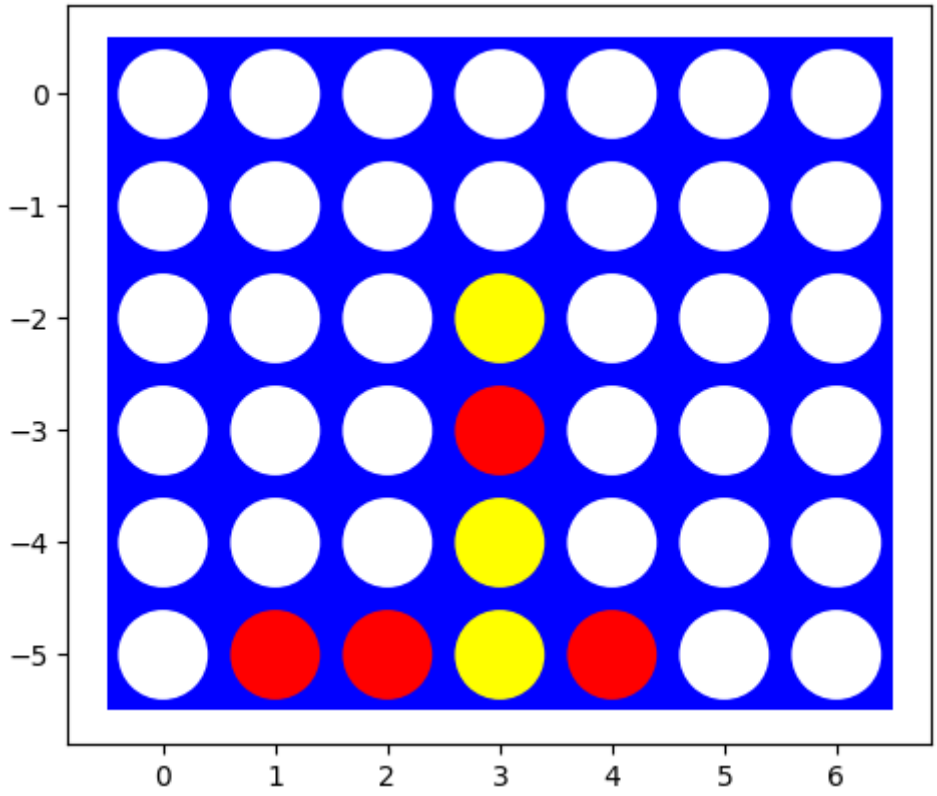

```
Enter your column: 3
```



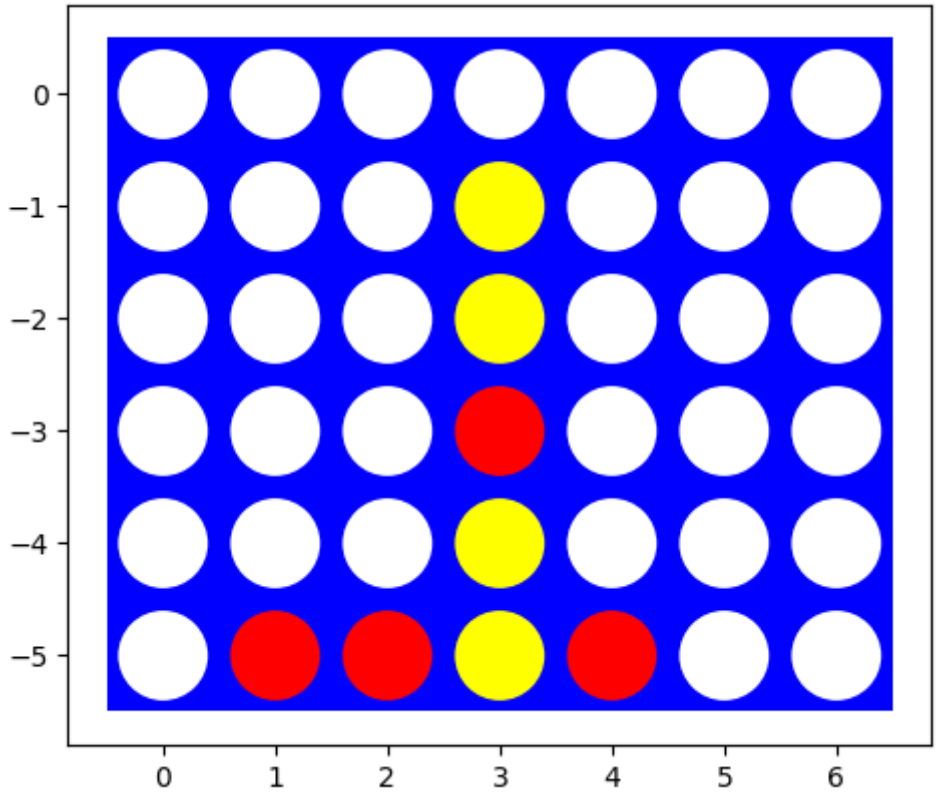

```
AI's move:
```



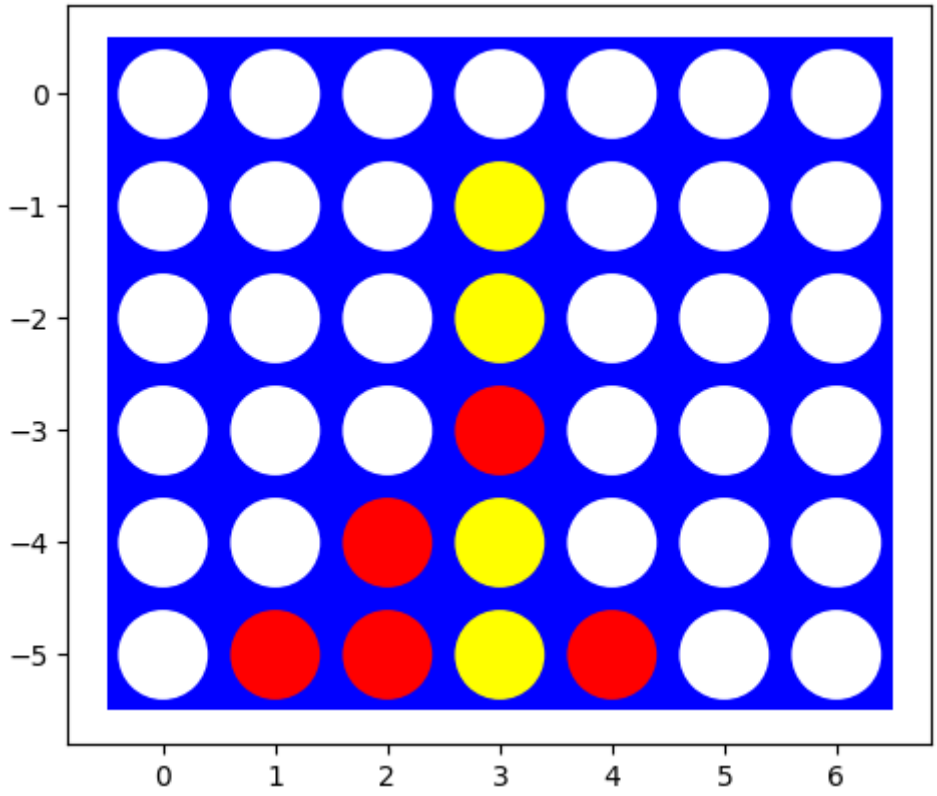

```
Enter your column: 2
```



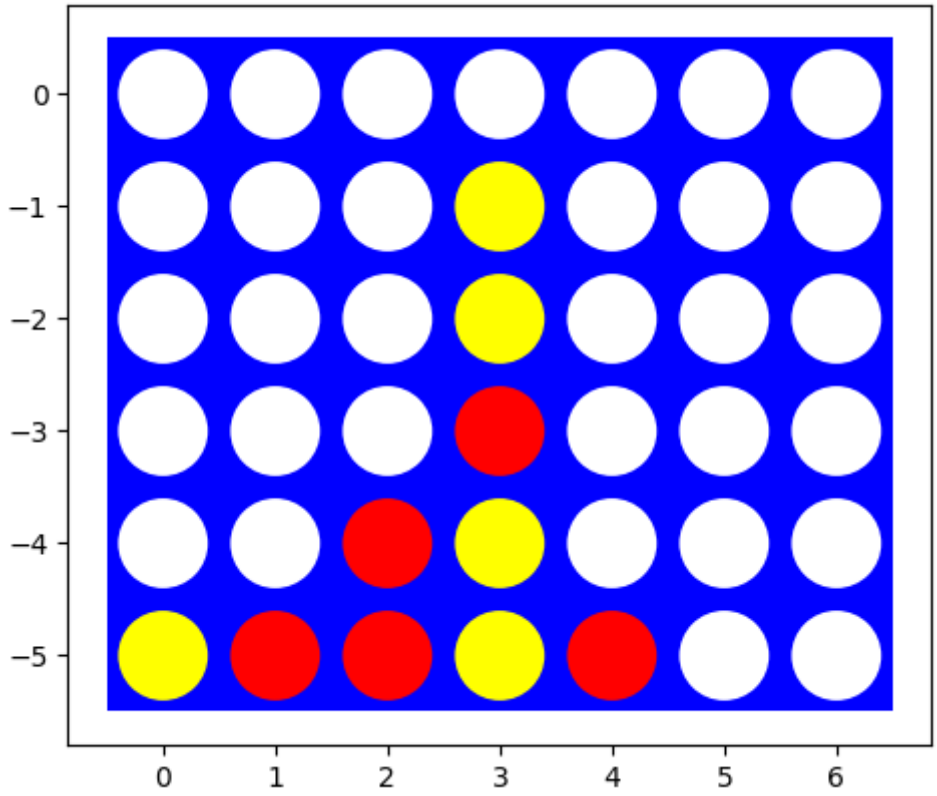

```
AI's move:
```



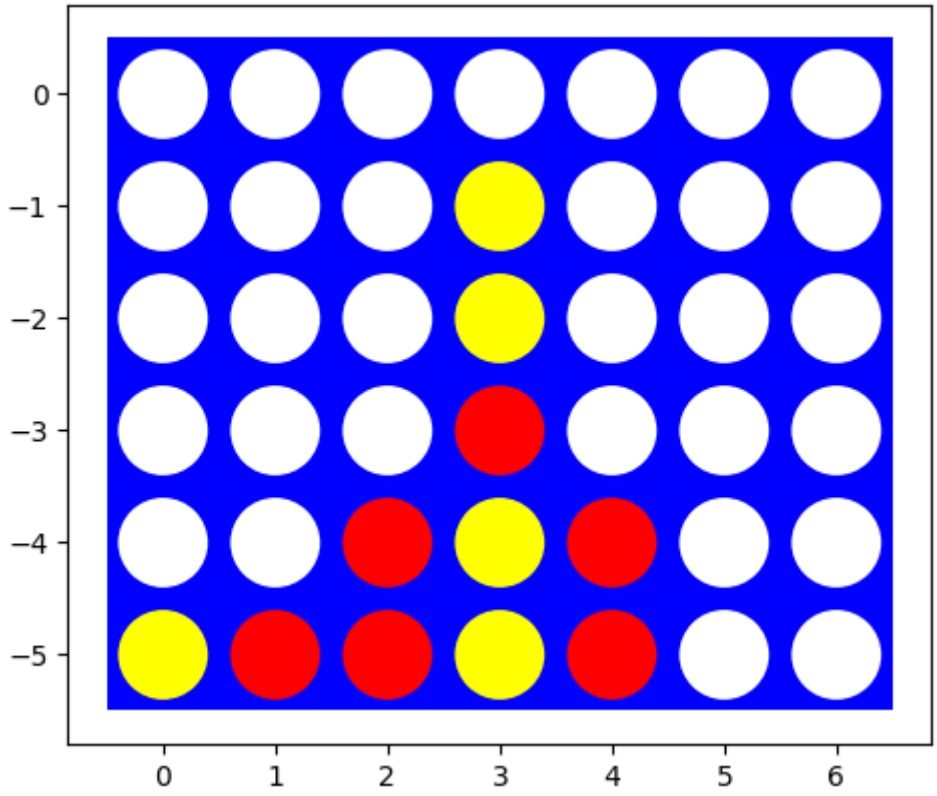

```
Enter your column: 2
```



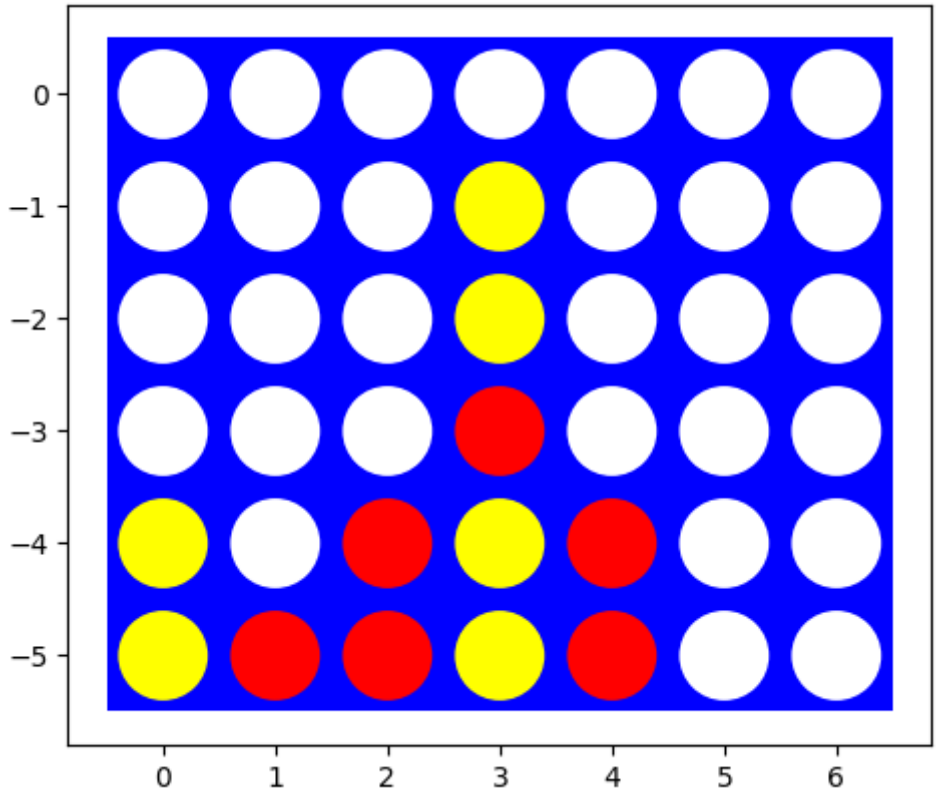

```
AI's move:
```



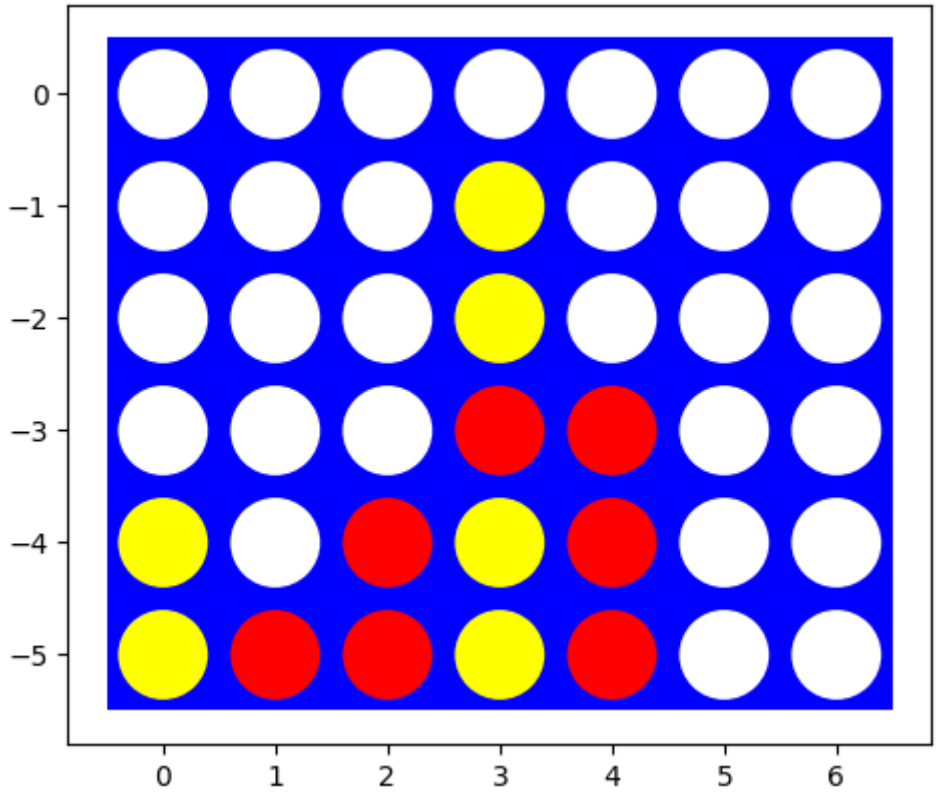

```
Enter your column: 4
```



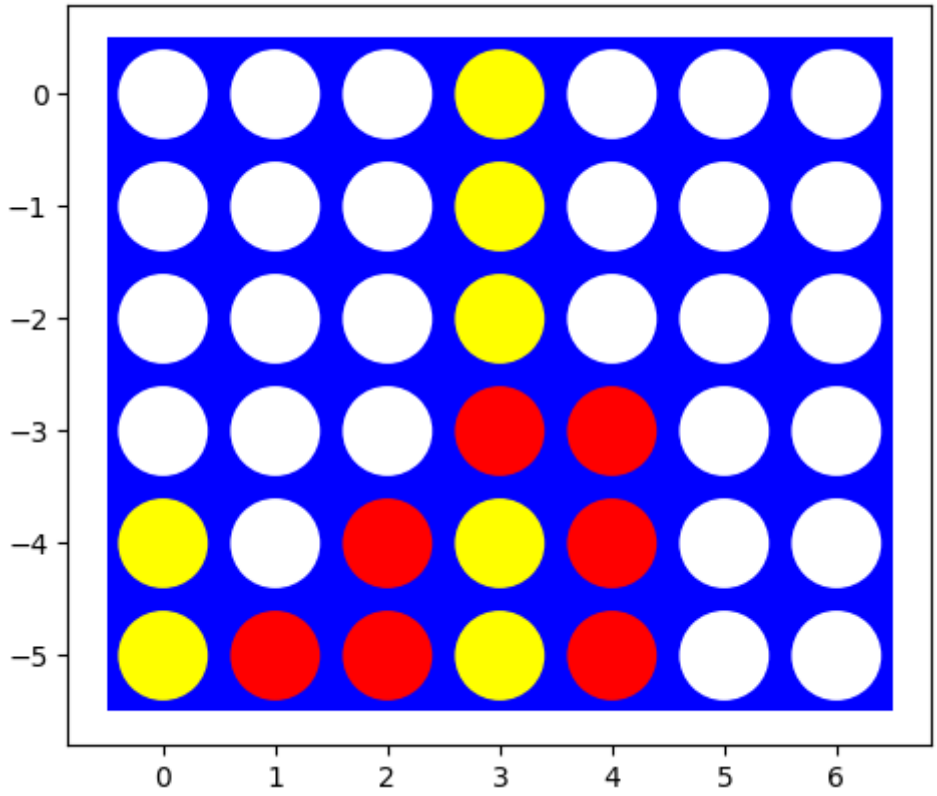

```
AI's move:
```



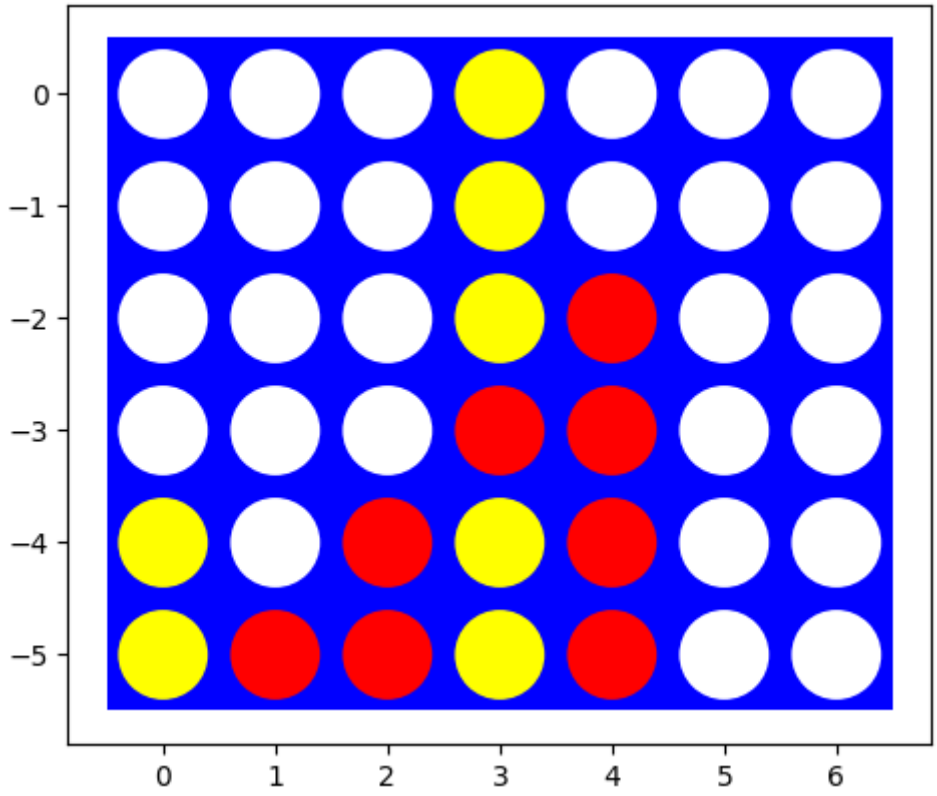

```
Enter your column: 4
```




```
AI's move:
```




```
Enter your column: 4
```



Experiment with the same manually created boards as above to check if the agent spots wining opportunities.

In [30]:

```
import numpy as np
import math
import copy

rows, cols = 6, 7

def heuristic(board, player):
```

```
    np_board = np.array(board)  # Convert to NumPy array
    center_column = np_board[:, len(np_board[0]) // 2]
    player_pieces_in_center = center_column.tolist().count(player)
    return player_pieces_in_center

def heuristic_alpha_beta_search(board, player, depth, alpha, beta):
    def max_value(board, player, depth, alpha, beta):
        if depth == 0 or is_terminal_state(board):
            return heuristic(board, player)

        value = -math.inf
        valid_moves = valid_actions_check(board)

        for move in valid_moves:
            new_board = copy.deepcopy(board)
            make_move(new_board, move, player)
            value = max(value, min_value(new_board, player, depth - 1, alpha, beta))

            alpha = max(alpha, value)
            if alpha >= beta:
                break

        return value

    def min_value(board, player, depth, alpha, beta):
        if depth == 0 or is_terminal_state(board):
            return heuristic(board, player)

        value = math.inf
        valid_moves = valid_actions_check(board)

        for move in valid_moves:
            new_board = copy.deepcopy(board)
            make_move(new_board, move, player)
            value = min(value, max_value(new_board, player, depth - 1, alpha, beta))

            beta = min(beta, value)
            if alpha >= beta:
                break

        return value

    best_move = None
    max_score = -math.inf
    valid_moves = valid_actions_check(board)

    for move in valid_moves:
        new_board = copy.deepcopy(board)
        make_move(new_board, move, player)
        score = min_value(new_board, player, depth - 1, alpha, beta)

        if score > max_score:
            max_score = score
            best_move = move

        alpha = max(alpha, max_score)
        if alpha >= beta:
            break

    return best_move

# Testing heuristic_alpha_beta_search with a sample board
test_board = [
    [0, 0, 0, 0, 0, 0, 0],
```

```
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0]
]

print(heuristic_alpha_beta_search(test_board, 1, depth=3, alpha=-math.inf, beta=math
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-30-af3c33423916> in <cell line: 86>()
     84 ]
     85
---> 86 print(heuristic_alpha_beta_search(test_board, 1, depth=3, alpha=-math.inf, be
ta=math.inf))

<ipython-input-30-af3c33423916> in heuristic_alpha_beta_search(board, player, depth,
alpha, beta)
     57     best_move = None
     58     max_score = -math.inf
---> 59     valid_moves = valid_actions_check(board)
     60
     61     for move in valid_moves:

TypeError: valid_actions_check() missing 1 required positional argument: 'col'
```

How long does it take to make a move? Start with a smaller board with 4 columns and make the board larger by adding columns.

In [33]:

```
import time

def measure_time_for_move(board, player, depth):
    start = time.time()
    _, _ = heuristic_alpha_beta_search(board, player, -math.inf, math.inf, depth)
    return time.time() - start

def main():
    for num_columns in range(4, 8):
        board = initialize_board(6, num_columns)
        #visualize(board)

        # Time measurement for Human move
        human_time = measure_time_for_move(board, 1, 1)
        print(f"Time for Human move on a {6}x{num_columns} board: {human_time:.5f} s

        # Time measurement for AI move
        ai_time = measure_time_for_move(board, 2, 3)  # Adjust depth based on your c
        print(f"Time for AI move on a {6}x{num_columns} board: {ai_time:.5f} seconds

if __name__ == "__main__":
    main()
```

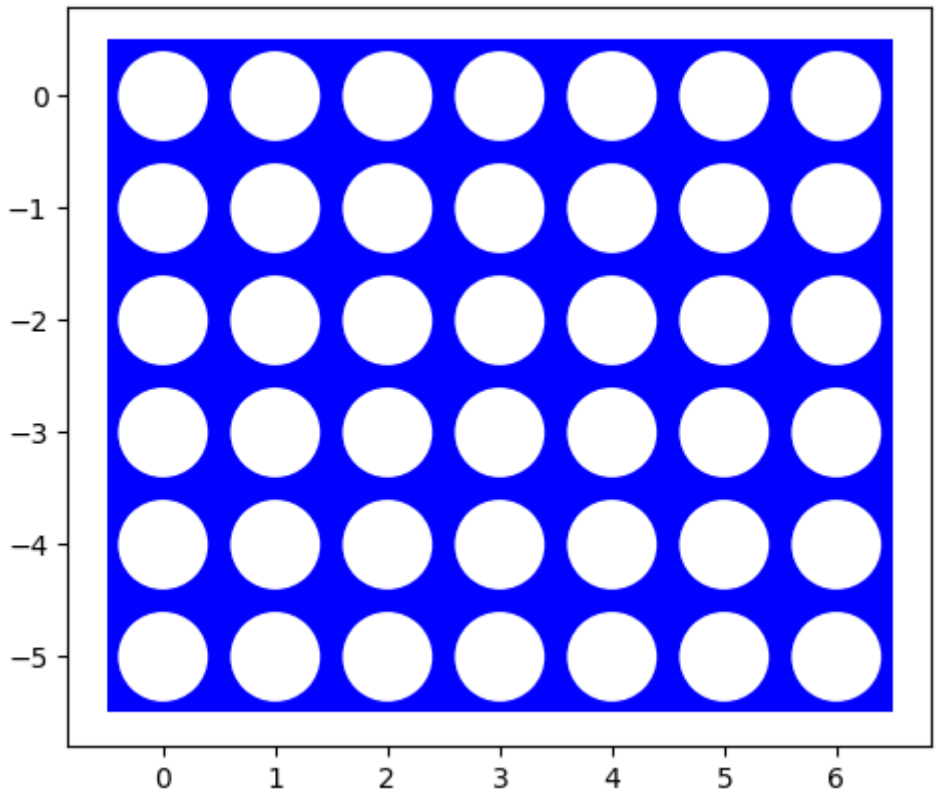
```
Time for Human move on a 6x4 board: 0.00231 seconds
Time for AI move on a 6x4 board: 0.04012 seconds
Time for Human move on a 6x5 board: 0.00149 seconds
Time for AI move on a 6x5 board: 0.03876 seconds
Time for Human move on a 6x6 board: 0.00152 seconds
Time for AI move on a 6x6 board: 0.03857 seconds
Time for Human move on a 6x7 board: 0.00618 seconds
Time for AI move on a 6x7 board: 0.04167 seconds
```

Playtime [5 points]

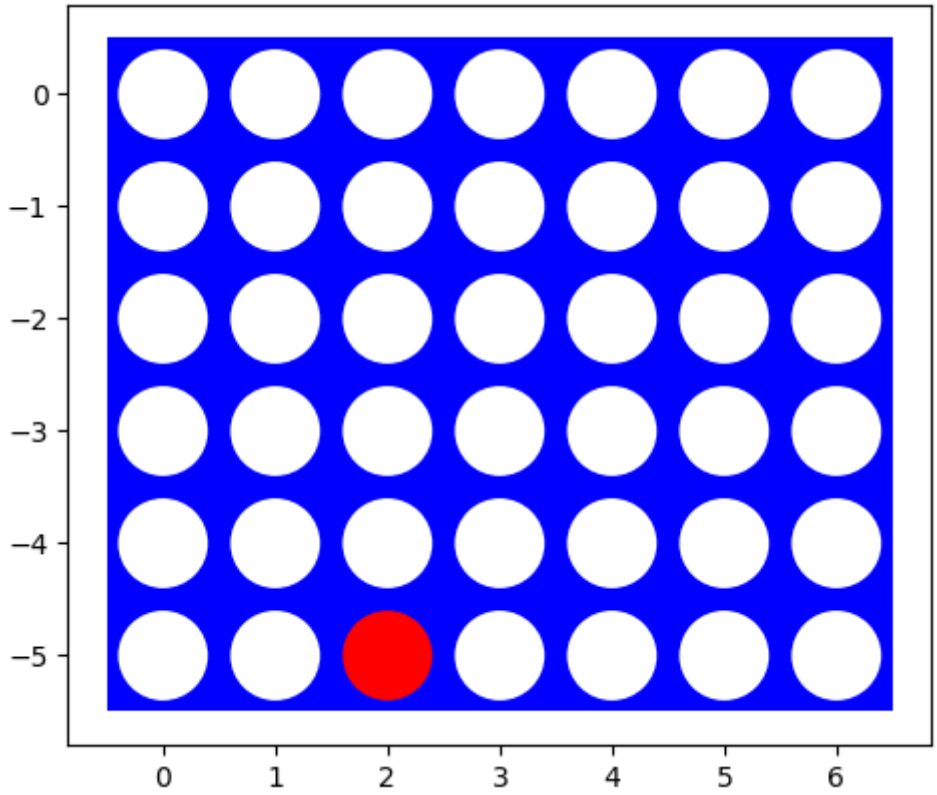
Let two heuristic search agents (different cutoff depth, different heuristic evaluation function) compete against each other on a reasonably sized board. Since there is no randomness, you only need to let them play once.

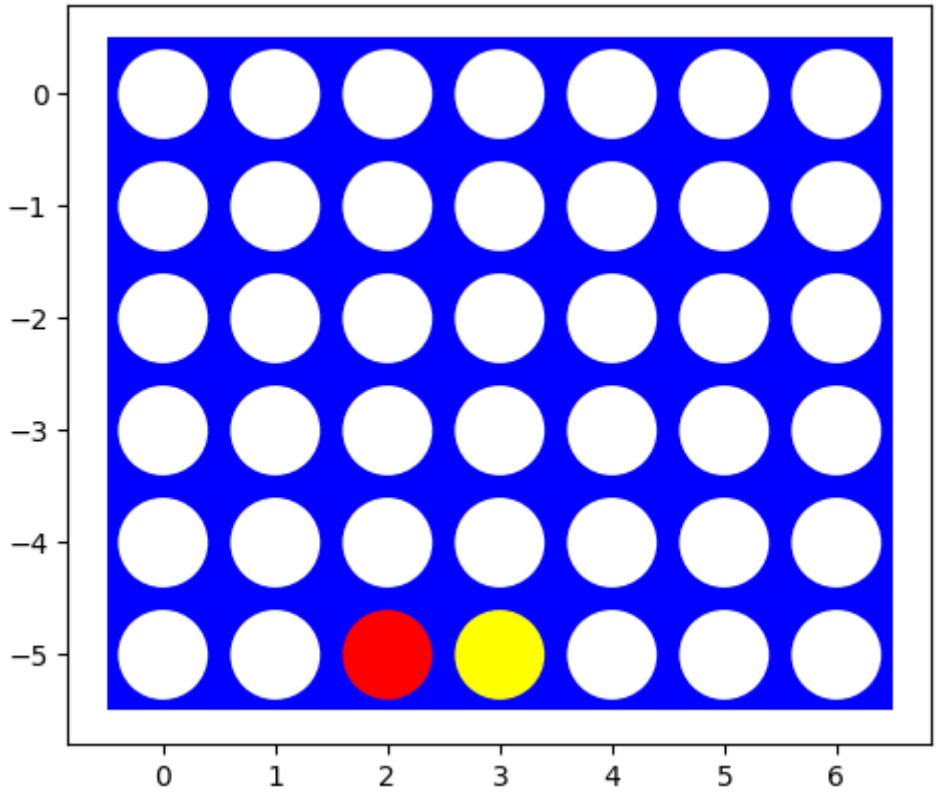
In [50]:

```
# Your code/ answer goes here.
game_over_check = False
turn = 0  # Human turn is first
difficulty = 0

board = initialize_board(6,7)
visualize(board)

while not game_over_check:
    if turn == 0:
        col = int(input("Enter the column:"))
        if valid_actions_check(board, col):
            row = get_row(board, col)
            update_board(board, row, col, 1)
            #print_board(board)
            if check_winner(board,1):
                print("YOU WIN CONGRATS!")
                game_over_check = True
            turn += 1
            visualize(board)
    elif turn == 1:
        func_score, best_col = minmax(board, True, -math.inf, math.inf, 3)
        if valid_actions_check(board, best_col):
            row = get_row(board, best_col)
            update_board(board, row, best_col, 2)
            #print_board(board)
            if check_winner(board, 2):
                print("AI WIN CONGRATS!")
                game_over_check = True
            turn -= 1
            visualize(board)
```

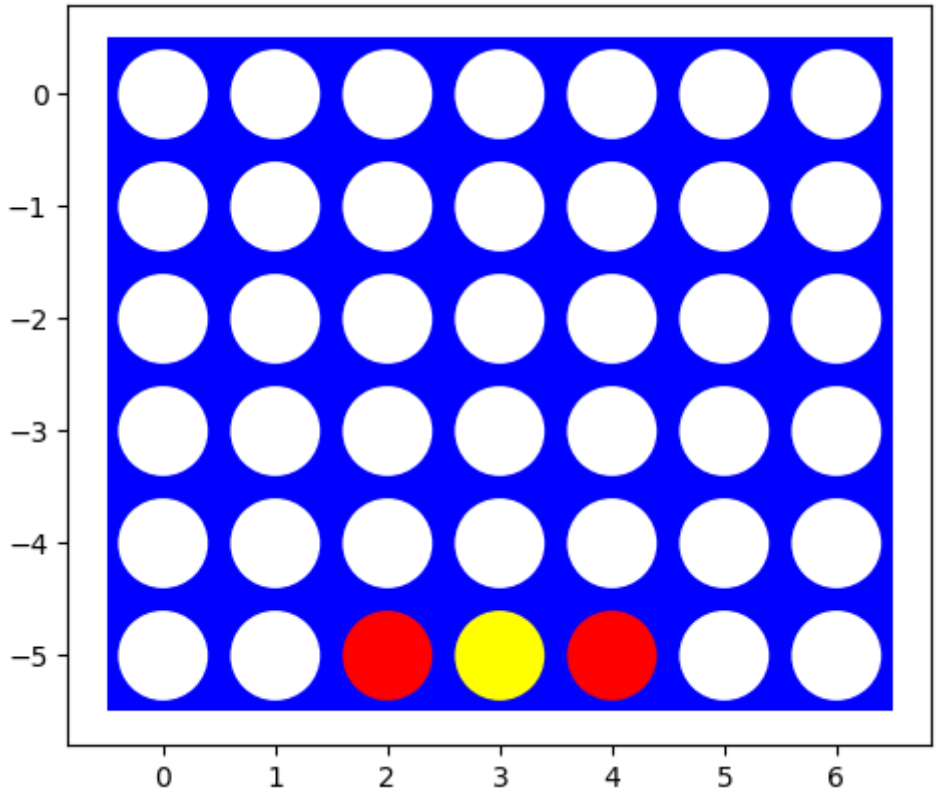


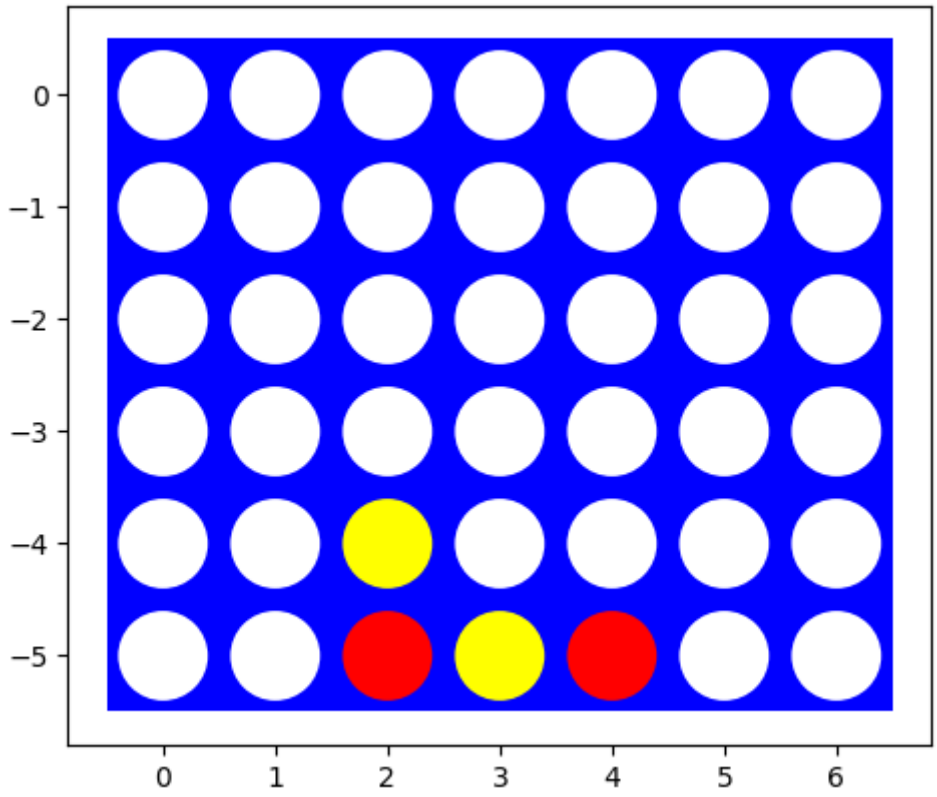
```
Enter the column:2
```



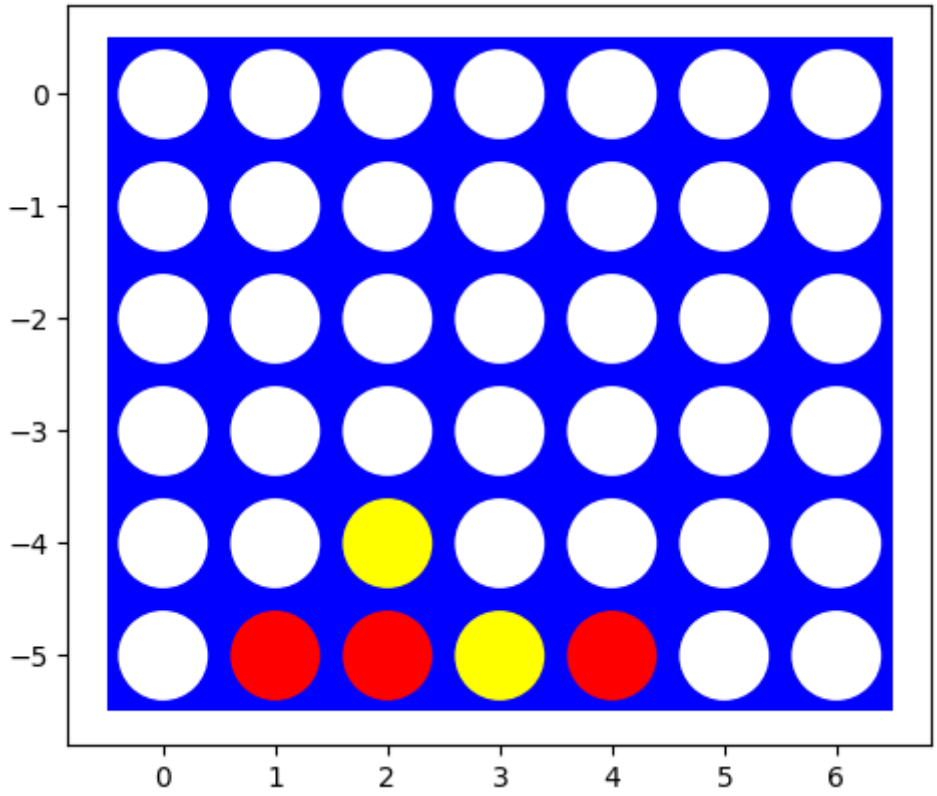


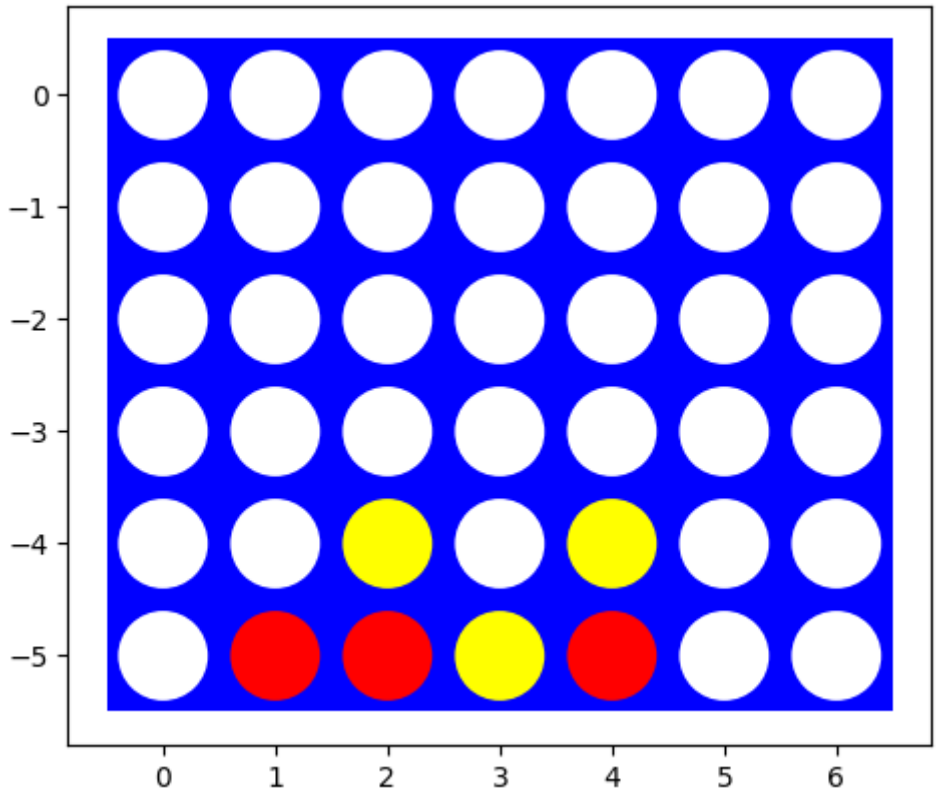

```
Enter the column:4
```



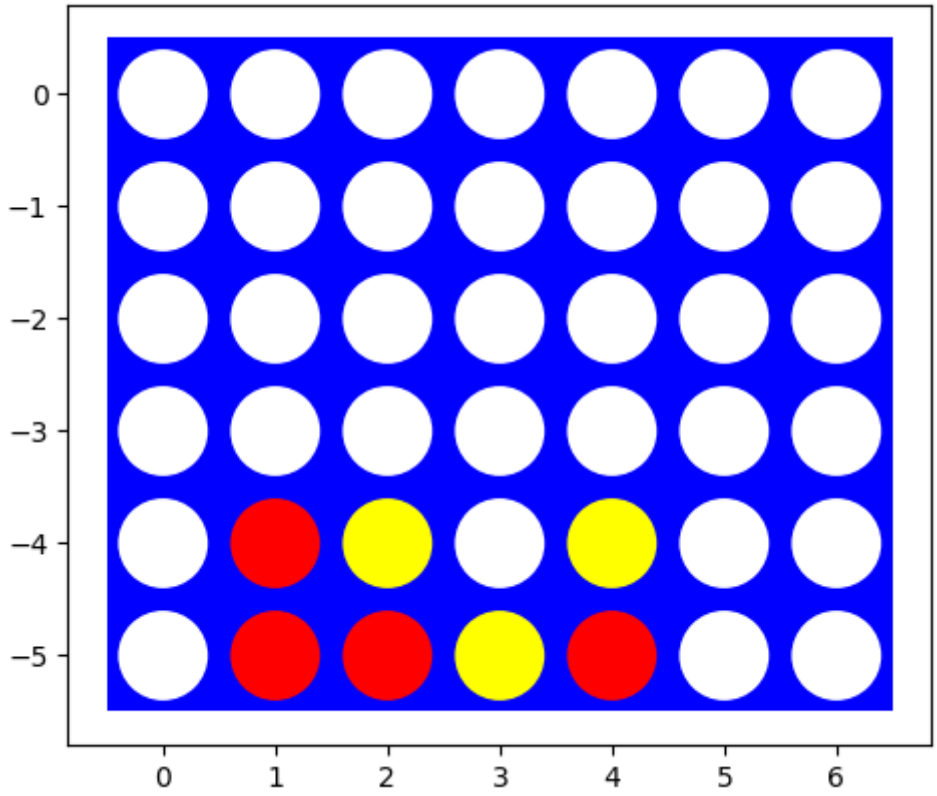


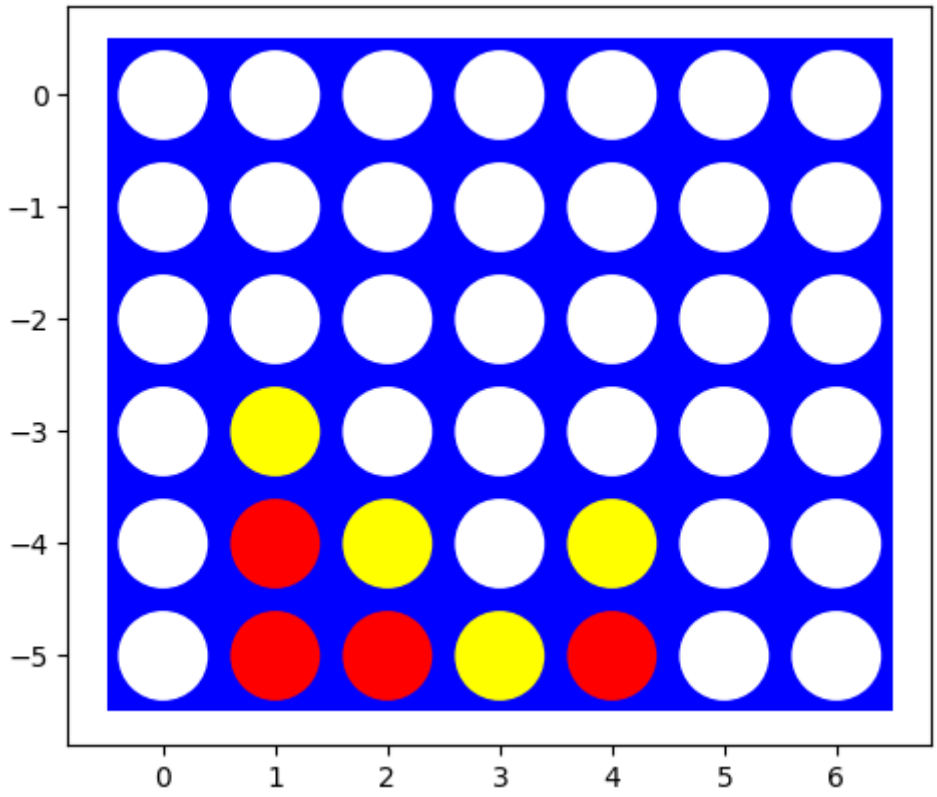

```
Enter the column:1
```



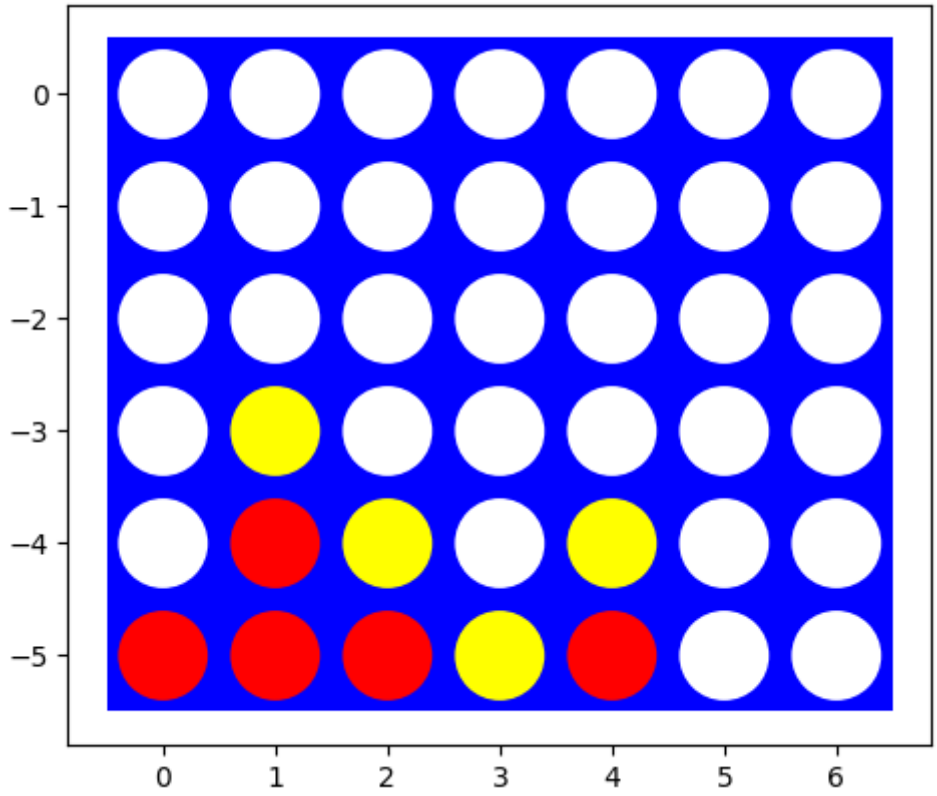


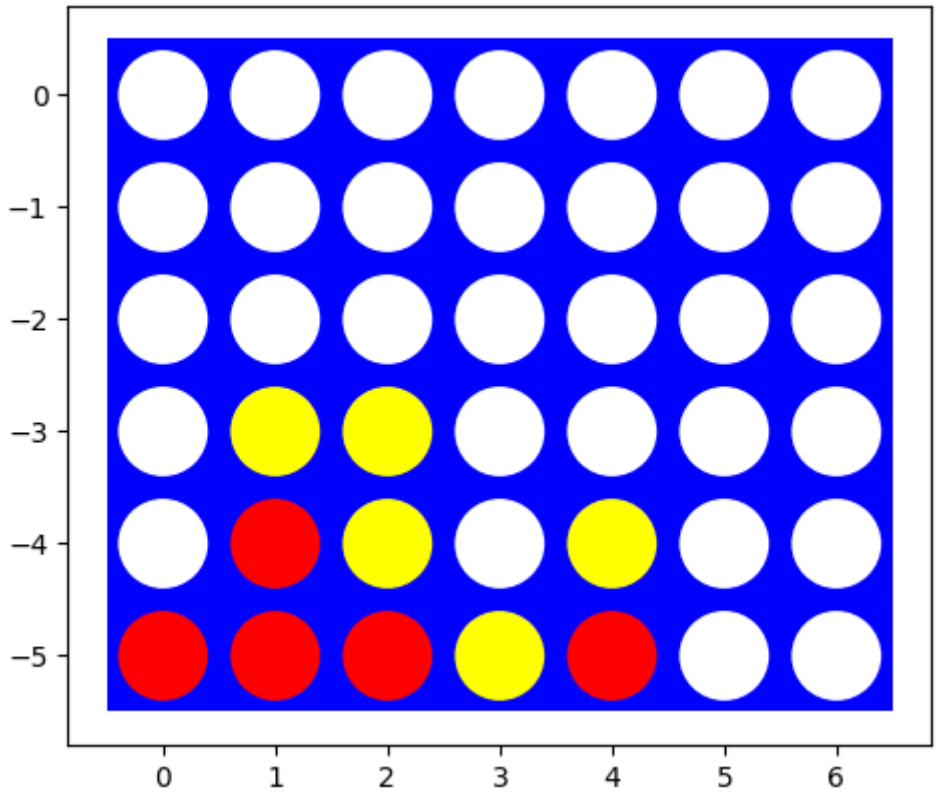

```
Enter the column:1
```



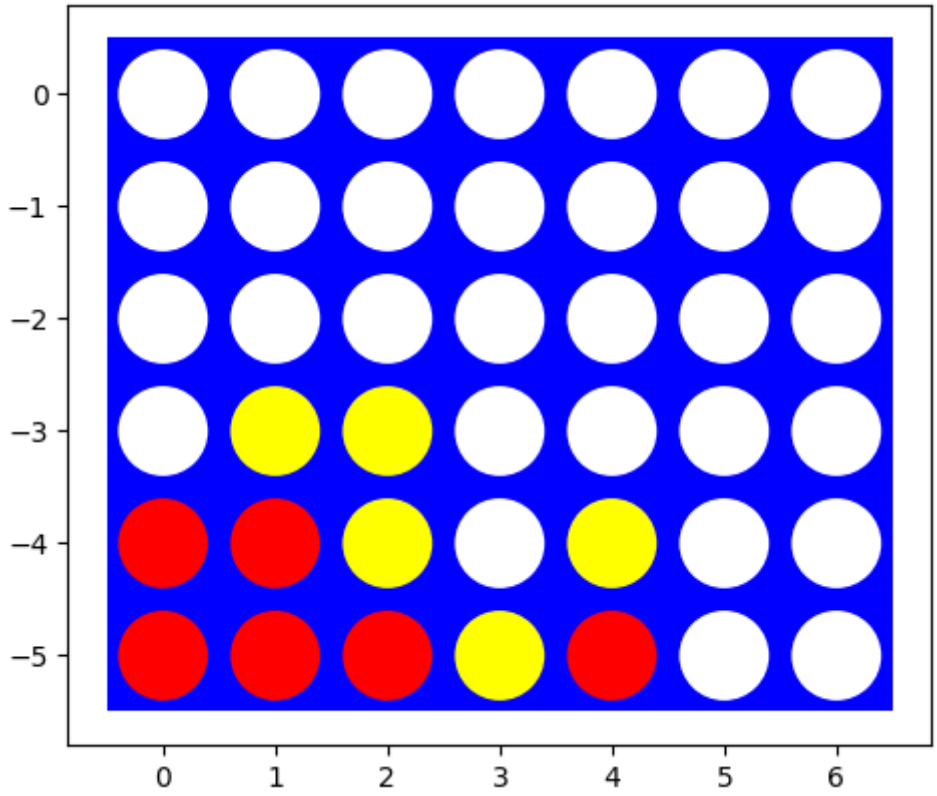


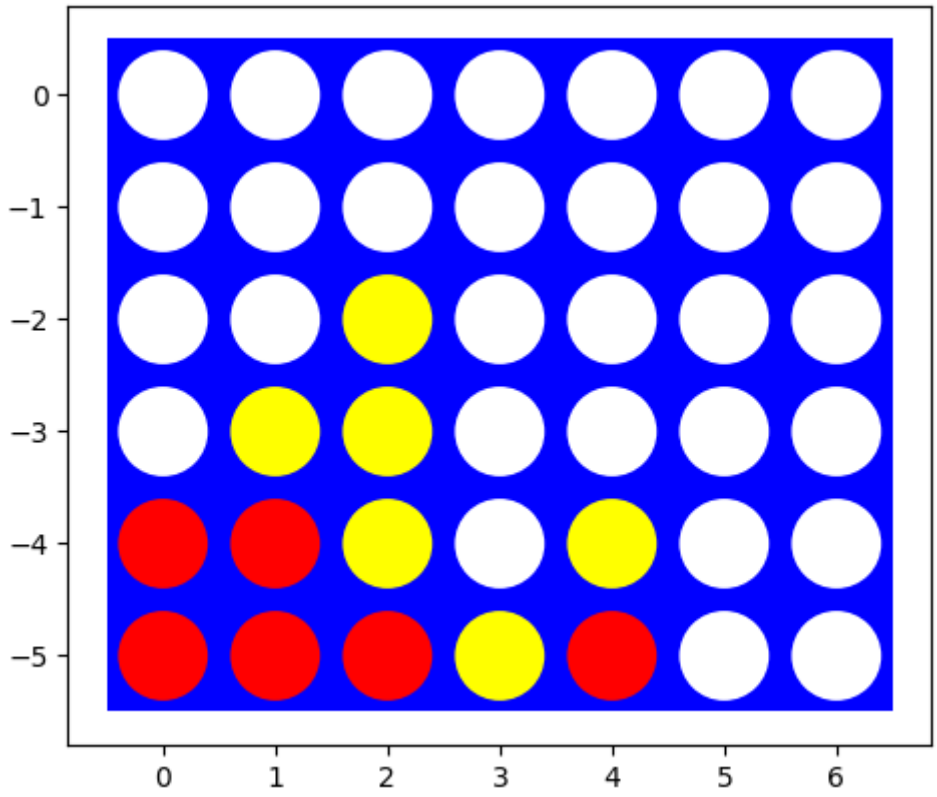

```
Enter the column:0
```



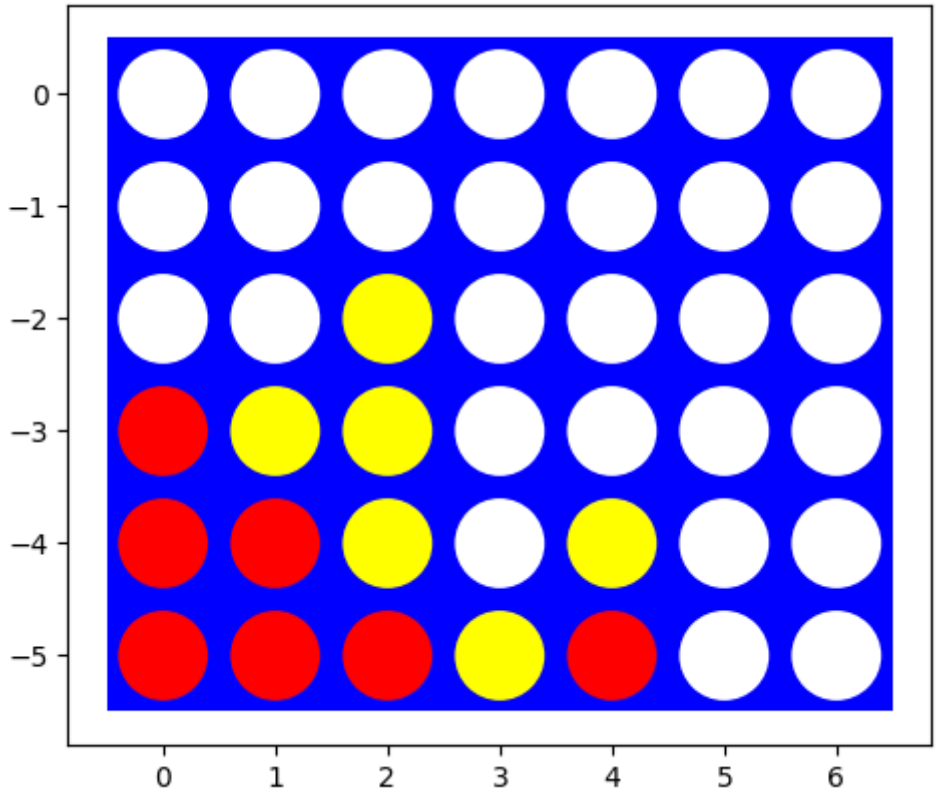



```
Enter the column:0
```

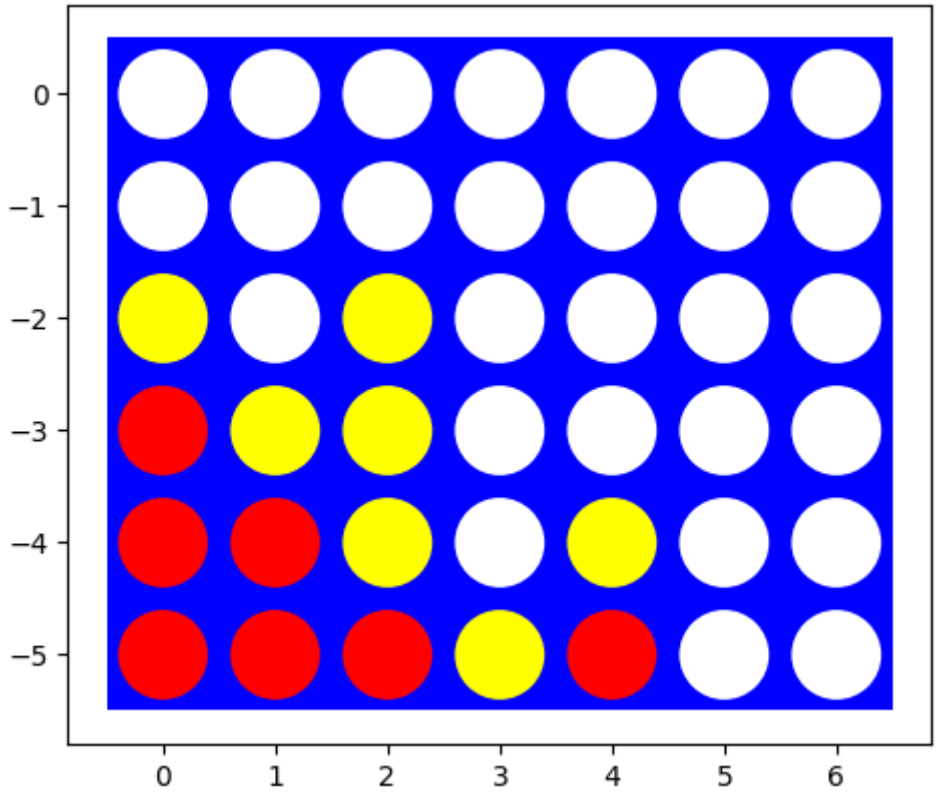





```
Enter the column:0
```




```
AI WIN CONGRATS!
```



Challenge task [+ 10 bonus point will be awarded separately]

Find another student and let your best agent play against the other student's best player. We will set up a class tournament on Canvas. This tournament will continue after the submission deadline.

In [ ]:

Graduate student advanced task: Pure Monte Carlo Search and Best First Move [10 point]

Undergraduate students: This is a bonus task you can attempt if you like [+10 bonus point].

Pure Monte Carlo Search

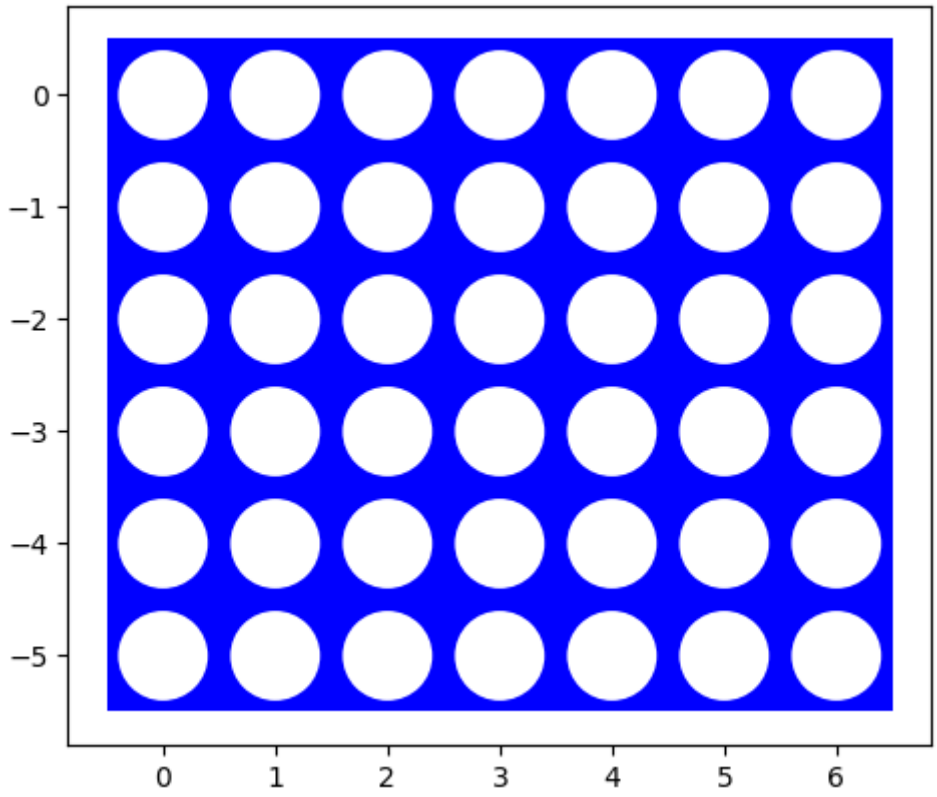
Implement Pure Monte Carlo Search and investigate how this search performs on the test boards that you have used above.

The pure_monte_carlo_search function performs Monte Carlo simulations to determine the best move for the AI player. It conducts random simulations for a specified number of iterations (default is 1000). In each simulation, random moves are made until a terminal state is reached, and the evaluate function is used to assign a score.

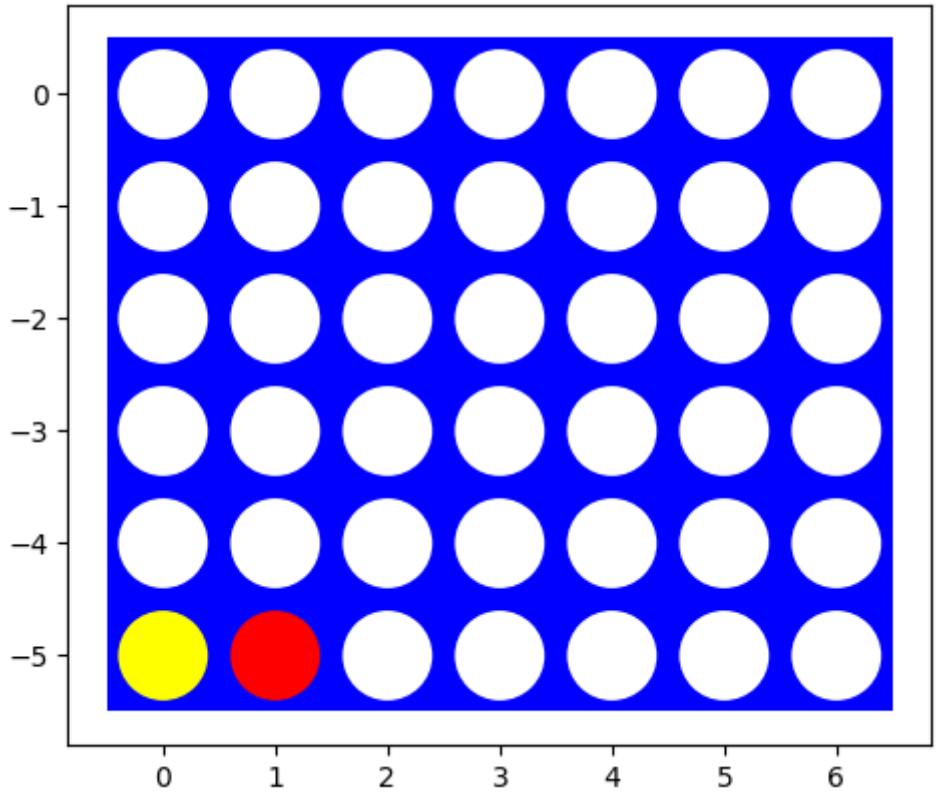
In [ ]:

```
import copy
def evaluate(board, player):
    # Simple evaluation: +100 for AI win, -100 for human win, 0 otherwise
    if check_winner(board, 2):
        return 100
    elif check_winner(board, 1):
        return -100
    else:
        return 0

def pure_monte_carlo_search(board, iterations=1000):
    player = 2  # AI player

    def random_simulation(board):
        board_copy = copy.deepcopy(board)
        while not is_terminal_state(board_copy):
            action = valid_actions(board_copy)
            random_move = random.choice(action)
            row = get_row(board_copy, random_move)
            update_board(board_copy, row, random_move, player % 2 + 1)  # Switch pla
        return evaluate(board_copy, player)

    root_valid_moves = valid_actions(board)

    if not root_valid_moves:
        return None

    move_values = {}

    for move in root_valid_moves:
        total_score = 0

        for _ in range(iterations):
            simulation_result = random_simulation(board)
            total_score += simulation_result

        average_score = total_score / iterations
        move_values[move] = average_score

    best_move = max(move_values, key=move_values.get)
    return best_move
```

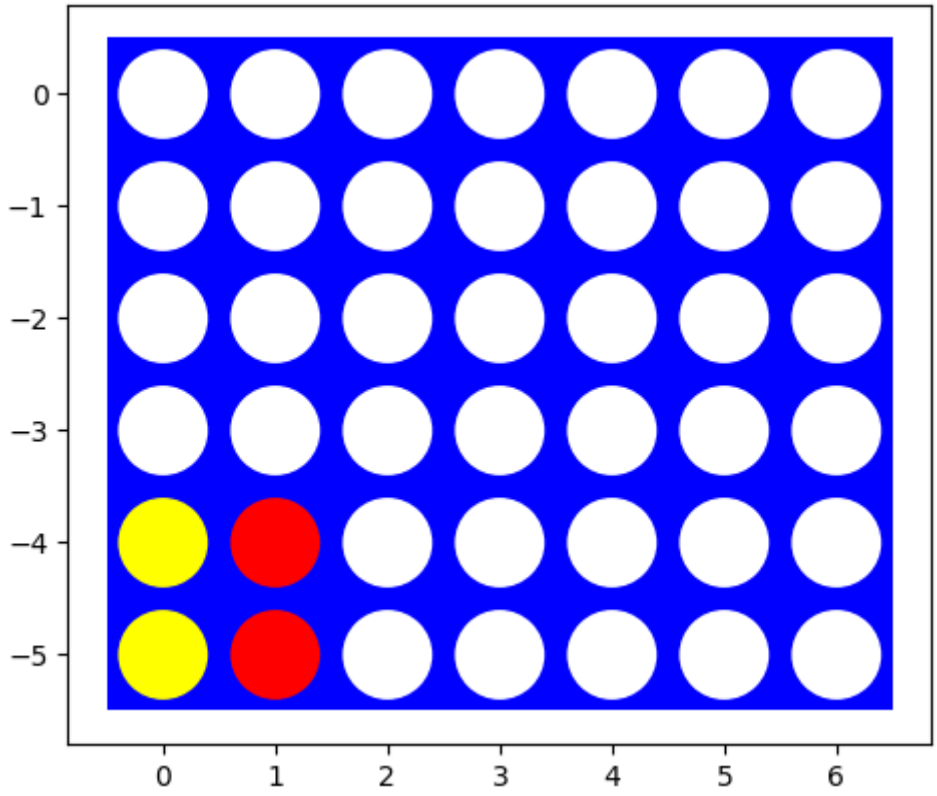
```python
if __name__ == "__main__":
    board = initialize_board(6,7)

    while not is_terminal_state(board):
        visualize(board)
        action = int(input("Enter your column: "))
        if valid_actions_check(board, action):
            row = get_row(board, action)
            update_board(board, row, action, 1)

        if is_terminal_state(board):
            break

        print("AI's move:")
        ai_move = pure_monte_carlo_search(board, 1000)
        ai_row = get_row(board, ai_move)
        update_board(board, ai_row, ai_move, 2)

    visualize(board)
    if check_winner(board, 1):
        print("You win!")
    elif check_winner(board, 2):
        print("AI wins!")
    else:
        print("It's a draw!")
```



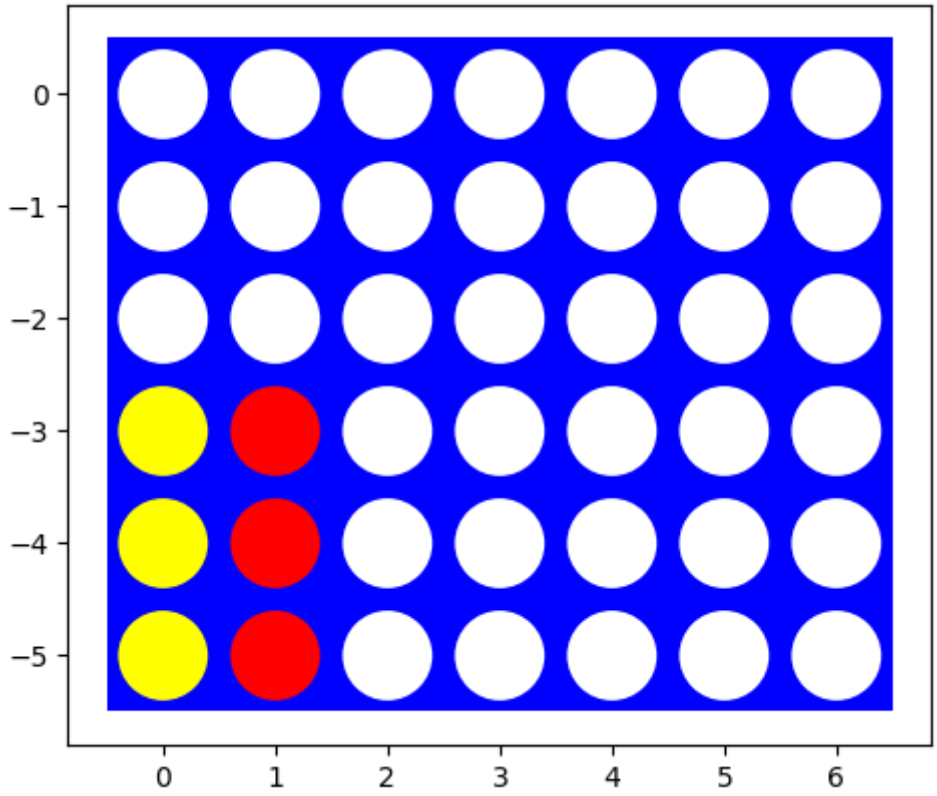

```
Enter your column: 1
AI's move:
```



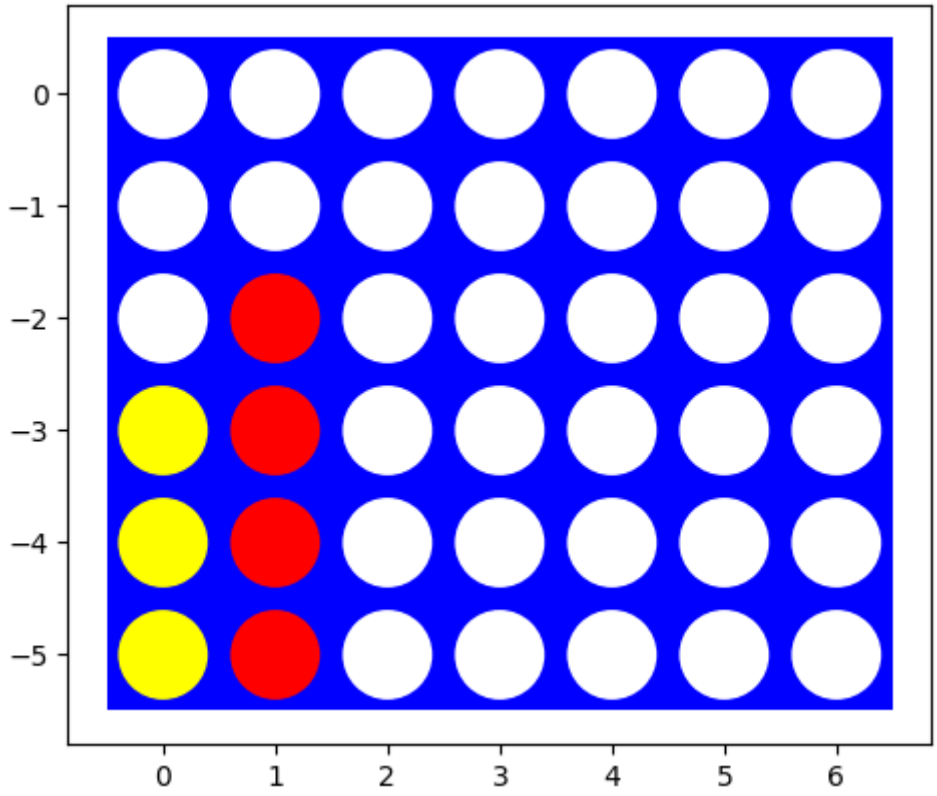

```
Enter your column: 1
AI's move:
```




```
Enter your column: 1
AI's move:
```




```
Enter your column: 1
```




```
You win!
```

Best First Move

Use Pure Monte Carlo Search to determine what the best first move is? Describe under what assumptions this is the "best" first move.

How to Use Pure Monte Carlo Search to Find the Best Initial Step:

1. Initialize the Board

2. Implement the Function of Monte Carlo Simulation: In this function, the game must be randomly executed from the current board state until a terminal condition is reached.
3. Choose Potential Moves: Determine every move (column) that is now allowed to be used to drop a ball on the board.
4. Replicate Motions: Use the Monte Carlo simulation program to determine the likely result of each playable move.
5. Assess Motions: Determine the average result (win, lose, or draw) for every move across all simulations.
6. Decide on the Best Action: The "best" first move is the one that yields the highest average outcome.