In [1]:
```
!pip install easyAI
```

```
Collecting easyAI
  Downloading easyAI-2.0.12-py3-none-any.whl (42 kB)
Requirement already satisfied: numpy in c:\users\smain\appdata\local\programs\pyt
hon\python310\lib\site-packages (from easyAI) (1.26.4)
Installing collected packages: easyAI
Successfully installed easyAI-2.0.12
```

```
WARNING: You are using pip version 21.2.3; however, version 25.0.1 is available.
You should consider upgrading via the 'C:\Users\smain\AppData\Local\Programs\Pyth
on\Python310\python.exe -m pip install --upgrade pip' command.
```

In [2]:
```
!pip install easyAI==2.0.12
```

```
Requirement already satisfied: easyAI==2.0.12 in c:\users\smain\appdata\local\pro
grams\python\python310\lib\site-packages (2.0.12)
Requirement already satisfied: numpy in c:\users\smain\appdata\local\programs\pyt
hon\python310\lib\site-packages (from easyAI==2.0.12) (1.26.4)
```

```
WARNING: You are using pip version 21.2.3; however, version 25.0.1 is available.
You should consider upgrading via the 'C:\Users\smain\AppData\Local\Programs\Pyth
on\Python310\python.exe -m pip install --upgrade pip' command.
```

In [3]:
```
# This is a variant of the Game of Bones recipe given in the easyAI library

from easyAI import TwoPlayersGame, id_solve, Human_Player, AI_Player
from easyAI.AI import TT

class LastCoinStanding(TwoPlayersGame):
    def __init__(self, players):
        # Define the players. Necessary parameter.
        self.players = players

        # Define who starts the game. Necessary parameter.
        self.nplayer = 1

        # Overall number of coins in the pile
        self.num_coins = 25

        # Define max number of coins per move
        self.max_coins = 4

    # Define possible moves
    def possible_moves(self):
        return [str(x) for x in range(1, self.max_coins + 1)]

    # Remove coins
    def make_move(self, move):
        self.num_coins -= int(move)

    # Did the opponent take the last coin?
    def win(self):
        return self.num_coins <= 0

    # Stop the game when somebody wins
    def is_over(self):
        return self.win()

    # Compute score
```

```
    def scoring(self):
        return 100 if self.win() else 0

    # Show number of coins remaining in the pile
    def show(self):
        print(self.num_coins, 'coins left in the pile')

if __name__ == "__main__":
    # Define the transposition table
    tt = TT()

    # Define the method
    LastCoinStanding.ttentry = lambda self: self.num_coins

    # Solve the game
    result, depth, move = id_solve(LastCoinStanding,
            range(2, 20), win_score=100, tt=tt)
    print(result, depth, move)

    # Start the game
    game = LastCoinStanding([AI_Player(tt), Human_Player()])
    game.play()
```

```
d:2, a:0, m:1
d:3, a:0, m:1
d:4, a:0, m:1
d:5, a:0, m:1
d:6, a:0, m:1
d:7, a:0, m:1
d:8, a:0, m:1
d:9, a:0, m:1
d:10, a:100, m:4
1 10 4
25 coins left in the pile

Move #1: player 1 plays 4 :
21 coins left in the pile

Move #2: player 2 plays 2 :
19 coins left in the pile

Move #3: player 1 plays 3 :
16 coins left in the pile

Move #4: player 2 plays 1 :
15 coins left in the pile

Move #5: player 1 plays 4 :
11 coins left in the pile

Move #6: player 2 plays 2 :
9 coins left in the pile

Move #7: player 1 plays 3 :
6 coins left in the pile

Move #8: player 2 plays 3 :
3 coins left in the pile

Move #9: player 1 plays 2 :
1 coins left in the pile

Move #10: player 2 plays 3 :
-2 coins left in the pile
```

Le bloc de code précédent implémente une variante du jeu "Last Coin Standing" en utilisant la bibliothèque easyAI, qui permet de créer et résoudre des jeux à deux joueurs avec de l'intelligence artificielle.

Le jeu commence avec 25 pièces, et à chaque tour, un joueur peut retirer entre 1 et 4 pièces. Le joueur qui prend la dernière pièce perd la partie. La classe LastCoinStanding hérite de TwoPlayersGame et définit les règles du jeu, y compris les coups possibles, la condition de victoire, et l'affichage de l'état du jeu.

Une table de transposition (TT) est utilisée pour mémoriser les états du jeu et optimiser l'IA. Avant de commencer, id_solve() précalcule la meilleure stratégie pour l'IA en évaluant les coups gagnants à différentes profondeurs. Ensuite, une partie est lancée entre l'IA et un joueur humain, où l'IA applique l'algorithme de recherche de la meilleure stratégie.

Ce programme illustre l'application des algorithmes d'IA dans les jeux de stratégie, combinant recherche de solutions optimales et jeu interactif contre une machine.

In [4]:

```python
# This is a variant of the Tic Tac Toe recipe given in the easyAI library

from easyAI import TwoPlayersGame, AI_Player, Negamax
from easyAI.Player import Human_Player

class GameController(TwoPlayersGame):
    def __init__(self, players):
        # Define the players
        self.players = players

        # Define who starts the game
        self.nplayer = 1

        # Define the board
        self.board = [0] * 9

    # Define possible moves
    def possible_moves(self):
        return [a + 1 for a, b in enumerate(self.board) if b == 0]

    # Make a move
    def make_move(self, move):
        self.board[int(move) - 1] = self.nplayer

    # Does the opponent have three in a line?
    def loss_condition(self):
        possible_combinations = [[1,2,3], [4,5,6], [7,8,9],
            [1,4,7], [2,5,8], [3,6,9], [1,5,9], [3,5,7]]

        return any([all([(self.board[i-1] == self.nopponent)
                for i in combination]) for combination in possible_combinations])

    # Check if the game is over
    def is_over(self):
        return (self.possible_moves() == []) or self.loss_condition()

    # Show current position
    def show(self):
        print('\n'+'\n'.join([' '.join([['.', 'O', 'X'][self.board[3*j + i]]
                for i in range(3)]) for j in range(3)]))

    # Compute the score
    def scoring(self):
        return -100 if self.loss_condition() else 0

if __name__ == "__main__":
    # Define the algorithm
    algorithm = Negamax(7)

    # Start the game
    GameController([Human_Player(), AI_Player(algorithm)]).play()
```

```
. . .
. . .
. . .

Move #1: player 1 plays 1 :

O . .
. . .
. . .

Move #2: player 2 plays 5 :

O . .
. X .
. . .

Move #3: player 1 plays 3 :

O . O
. X .
. . .

Move #4: player 2 plays 2 :

O X O
. X .
. . .

Move #5: player 1 plays 8 :

O X O
. X .
. O .

Move #6: player 2 plays 4 :

O X O
X X .
. O .

Move #7: player 1 plays 6 :

O X O
X X O
. O .

Move #8: player 2 plays 9 :

O X O
X X O
. O X

Move #9: player 1 plays 7 :

O X O
X X O
O O X
```

Le bloc de code précédent implémente une version du jeu Tic-Tac-Toe en utilisant la bibliothèque easyAI, qui permet de créer et jouer à des jeux avec une intelligence

artificielle.

La classe GameController hérite de TwoPlayersGame et définit les règles du jeu, dont la gestion du plateau (une liste de 9 cases), les coups possibles, et la mise à jour du plateau après chaque coup. Un joueur gagne s'il aligne trois symboles identiques selon des combinaisons prédéfinies.

L'IA utilise l'algorithme Negamax, une variante de Minimax optimisée pour les jeux à somme nulle. L'algorithme explore les coups possibles jusqu'à une profondeur de 7 niveaux, en attribuant un score négatif si l'adversaire gagne.

Le programme affiche l'état du jeu après chaque tour et vérifie si la partie est terminée (victoire ou grille pleine). Ensuite, une partie commence entre un joueur humain et l'IA, où l'IA applique Negamax pour jouer de manière optimale.

Ce code démontre l'application des algorithmes d'IA dans les jeux de stratégie, en permettant une interaction directe avec une intelligence artificielle compétitive.

In [5]:
```
# This is a variant of the Connect Four recipe given in the easyAI library

import numpy as np
from easyAI import TwoPlayersGame, Human_Player, AI_Player, \
        Negamax, SSS

class GameController(TwoPlayersGame):
    def __init__(self, players, board = None):
        # Define the players
        self.players = players

        # Define the configuration of the board
        self.board = board if (board != None) else (
            np.array([[0 for i in range(7)] for j in range(6)]))

        # Define who starts the game
        self.nplayer = 1

        # Define the positions
        self.pos_dir = np.array([[[i, 0], [0, 1]] for i in range(6)] +
                    [[[0, i], [1, 0]] for i in range(7)] +
                    [[[i, 0], [1, 1]] for i in range(1, 3)] +
                    [[[0, i], [1, 1]] for i in range(4)] +
                    [[[i, 6], [1, -1]] for i in range(1, 3)] +
                    [[[0, i], [1, -1]] for i in range(3, 7)])

    # Define possible moves
    def possible_moves(self):
        return [i for i in range(7) if (self.board[:, i].min() == 0)]

    # Define how to make the move
    def make_move(self, column):
        line = np.argmin(self.board[:, column] != 0)
        self.board[line, column] = self.nplayer

    # Show the current status
    def show(self):
```

```
        print('\n' + '\n'.join(
                ['0 1 2 3 4 5 6', 13 * '-'] +
                [' '.join([['.', 'O', 'X'][self.board[5 - j][i]]
                for i in range(7)]) for j in range(6)]))

    # Define what a loss_condition looks like
    def loss_condition(self):
        for pos, direction in self.pos_dir:
            streak = 0
            while (0 <= pos[0] <= 5) and (0 <= pos[1] <= 6):
                if self.board[pos[0], pos[1]] == self.nopponent:
                    streak += 1
                    if streak == 4:
                        return True
                else:
                    streak = 0

                pos = pos + direction

        return False

    # Check if the game is over
    def is_over(self):
        return (self.board.min() > 0) or self.loss_condition()

    # Compute the score
    def scoring(self):
        return -100 if self.loss_condition() else 0

if __name__ == '__main__':
    # Define the algorithms that will be used
    algo_neg = Negamax(5)
    algo_sss = SSS(5)

    # Start the game
    game = GameController([AI_Player(algo_neg), AI_Player(algo_sss)])
    game.play()

    # Print the result
    if game.loss_condition():
        print('\nPlayer', game.nopponent, 'wins.')
    else:
        print("\nIt's a draw.")
```

```
0 1 2 3 4 5 6
-------------
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .

Move #1: player 1 plays 0 :

0 1 2 3 4 5 6
-------------
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
O . . . . . .

Move #2: player 2 plays 0 :

0 1 2 3 4 5 6
-------------
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
X . . . . . .
O . . . . . .

Move #3: player 1 plays 0 :

0 1 2 3 4 5 6
-------------
. . . . . . .
. . . . . . .
. . . . . . .
O . . . . . .
X . . . . . .
O . . . . . .

Move #4: player 2 plays 0 :

0 1 2 3 4 5 6
-------------
. . . . . . .
. . . . . . .
X . . . . . .
O . . . . . .
X . . . . . .
O . . . . . .

Move #5: player 1 plays 0 :

0 1 2 3 4 5 6
-------------
. . . . . . .
O . . . . . .
X . . . . . .
```

```
O . . . . . .
X . . . . . .
O . . . . . .

Move #6: player 2 plays 0 :

0 1 2 3 4 5 6
-------------
X . . . . . .
O . . . . . .
X . . . . . .
O . . . . . .
X . . . . . .
O . . . . . .

Move #7: player 1 plays 1 :

0 1 2 3 4 5 6
-------------
X . . . . . .
O . . . . . .
X . . . . . .
O . . . . . .
X . . . . . .
O O . . . . .

Move #8: player 2 plays 1 :

0 1 2 3 4 5 6
-------------
X . . . . . .
O . . . . . .
X . . . . . .
O . . . . . .
X X . . . . .
O O . . . . .

Move #9: player 1 plays 1 :

0 1 2 3 4 5 6
-------------
X . . . . . .
O . . . . . .
X . . . . . .
O O . . . . .
X X . . . . .
O O . . . . .

Move #10: player 2 plays 1 :

0 1 2 3 4 5 6
-------------
X . . . . . .
O . . . . . .
X X . . . . .
O O . . . . .
X X . . . . .
O O . . . . .

Move #11: player 1 plays 1 :
```

```
0 1 2 3 4 5 6
-------------
X . . . . . .
O O . . . . .
X X . . . . .
O O . . . . .
X X . . . . .
O O . . . . .

Move #12: player 2 plays 1 :

0 1 2 3 4 5 6
-------------
X X . . . . .
O O . . . . .
X X . . . . .
O O . . . . .
X X . . . . .
O O . . . . .

Move #13: player 1 plays 2 :

0 1 2 3 4 5 6
-------------
X X . . . . .
O O . . . . .
X X . . . . .
O O . . . . .
X X . . . . .
O O O . . . .

Move #14: player 2 plays 3 :

0 1 2 3 4 5 6
-------------
X X . . . . .
O O . . . . .
X X . . . . .
O O . . . . .
X X . . . . .
O O O X . . .

Move #15: player 1 plays 2 :

0 1 2 3 4 5 6
-------------
X X . . . . .
O O . . . . .
X X . . . . .
O O . . . . .
X X O . . . .
O O O X . . .

Move #16: player 2 plays 2 :

0 1 2 3 4 5 6
-------------
X X . . . . .
O O . . . . .
```

```
X X . . . . .
O O X . . . .
X X O . . . .
O O O X . . .

Move #17: player 1 plays 2 :

0 1 2 3 4 5 6
-------------
X X . . . . .
O O . . . . .
X X O . . . .
O O X . . . .
X X O . . . .
O O O X . . .

Move #18: player 2 plays 2 :

0 1 2 3 4 5 6
-------------
X X . . . . .
O O X . . . .
X X O . . . .
O O X . . . .
X X O . . . .
O O O X . . .

Move #19: player 1 plays 2 :

0 1 2 3 4 5 6
-------------
X X O . . . .
O O X . . . .
X X O . . . .
O O X . . . .
X X O . . . .
O O O X . . .

Move #20: player 2 plays 3 :

0 1 2 3 4 5 6
-------------
X X O . . . .
O O X . . . .
X X O . . . .
O O X . . . .
X X O X . . .
O O O X . . .

Move #21: player 1 plays 4 :

0 1 2 3 4 5 6
-------------
X X O . . . .
O O X . . . .
X X O . . . .
O O X . . . .
X X O X . . .
O O O X O . .
```

```
Move #22: player 2 plays 3 :

0 1 2 3 4 5 6
-------------
X X O . . . .
O O X . . . .
X X O . . . .
O O X X . . .
X X O X . . .
O O O X O . .

Move #23: player 1 plays 3 :

0 1 2 3 4 5 6
-------------
X X O . . . .
O O X . . . .
X X O O . . .
O O X X . . .
X X O X . . .
O O O X O . .

Move #24: player 2 plays 3 :

0 1 2 3 4 5 6
-------------
X X O . . . .
O O X X . . .
X X O O . . .
O O X X . . .
X X O X . . .
O O O X O . .

Move #25: player 1 plays 3 :

0 1 2 3 4 5 6
-------------
X X O O . . .
O O X X . . .
X X O O . . .
O O X X . . .
X X O X . . .
O O O X O . .

Move #26: player 2 plays 4 :

0 1 2 3 4 5 6
-------------
X X O O . . .
O O X X . . .
X X O O . . .
O O X X . . .
X X O X X . .
O O O X O . .

Move #27: player 1 plays 4 :

0 1 2 3 4 5 6
-------------
X X O O . . .
```

```
O O X X . . .
X X O O . . .
O O X X O . .
X X O X X . .
O O O X O . .

Move #28: player 2 plays 4 :

0 1 2 3 4 5 6
-------------
X X O O . . .
O O X X . . .
X X O O X . .
O O X X O . .
X X O X X . .
O O O X O . .

Move #29: player 1 plays 4 :

0 1 2 3 4 5 6
-------------
X X O O . . .
O O X X O . .
X X O O X . .
O O X X O . .
X X O X X . .
O O O X O . .

Move #30: player 2 plays 4 :

0 1 2 3 4 5 6
-------------
X X O O X . .
O O X X O . .
X X O O X . .
O O X X O . .
X X O X X . .
O O O X O . .

Move #31: player 1 plays 5 :

0 1 2 3 4 5 6
-------------
X X O O X . .
O O X X O . .
X X O O X . .
O O X X O . .
X X O X X . .
O O O X O O .

Move #32: player 2 plays 5 :

0 1 2 3 4 5 6
-------------
X X O O X . .
O O X X O . .
X X O O X . .
O O X X O . .
X X O X X X .
O O O X O O .
```

```
Move #33: player 1 plays 5 :

0 1 2 3 4 5 6
-------------
X X O O X . .
O O X X O . .
X X O O X . .
O O X X O O .
X X O X X X .
O O O X O O .

Move #34: player 2 plays 5 :

0 1 2 3 4 5 6
-------------
X X O O X . .
O O X X O . .
X X O O X X .
O O X X O O .
X X O X X X .
O O O X O O .

Move #35: player 1 plays 6 :

0 1 2 3 4 5 6
-------------
X X O O X . .
O O X X O . .
X X O O X X .
O O X X O O .
X X O X X X .
O O O X O O O

Move #36: player 2 plays 6 :

0 1 2 3 4 5 6
-------------
X X O O X . .
O O X X O . .
X X O O X X .
O O X X O O .
X X O X X X X
O O O X O O O

Player 2 wins.
```

Le bloc de code précédent implémente une version du jeu Puissance 4 en utilisant la bibliothèque easyAI, permettant une intelligence artificielle compétitive.

La classe GameController, héritant de TwoPlayersGame, définit les règles du jeu. Elle initialise une grille de 6x7, où chaque joueur peut insérer un jeton dans une colonne. Les coups possibles sont les colonnes où il reste de la place. Le programme met ensuite à jour le plateau après chaque mouvement et affiche l'état actuel du jeu.

Une condition de victoire est détectée lorsqu'un joueur aligne 4 jetons consécutifs horizontalement, verticalement ou en diagonale. Le jeu s'arrête lorsqu'un joueur gagne ou si la grille est pleine (match nul).

Deux algorithmes d'IA sont utilisés pour l'optimisation : Negamax (5) et SSS (5). Ces algorithmes évaluent les mouvements possibles sur une profondeur de 5 coups pour déterminer la meilleure stratégie. La partie se joue automatiquement entre deux IA, et le programme affiche le gagnant ou un match nul.

Ce code illustre comment intégrer des algorithmes d'intelligence artificielle pour jouer à Puissance 4 de manière optimisée, rendant le jeu compétitif et stratégique.

In [6]:
```
# This is a variant of the Hexapawn recipe given in the easyAI library

from easyAI import TwoPlayersGame, AI_Player, \
        Human_Player, Negamax

class GameController(TwoPlayersGame):
    def __init__(self, players, size = (4, 4)):
        self.size = size
        num_pawns, len_board = size
        p = [[(i, j) for j in range(len_board)] \
                for i in [0, num_pawns - 1]]

        for i, d, goal, pawns in [(0, 1, num_pawns - 1,
                p[0]), (1, -1, 0, p[1])]:
            players[i].direction = d
            players[i].goal_line = goal
            players[i].pawns = pawns

        # Define the players
        self.players = players

        # Define who starts first
        self.nplayer = 1

        # Define the alphabets
        self.alphabets = 'ABCDEFGHIJ'

        # Convert B4 to (1, 3)
        self.to_tuple = lambda s: (self.alphabets.index(s[0]),
                int(s[1:]) - 1)

        # Convert (1, 3) to B4
        self.to_string = lambda move: ' '.join([self.alphabets[
                move[i][0]] + str(move[i][1] + 1)
                for i in (0, 1)])

    # Define the possible moves
    def possible_moves(self):
        moves = []
        opponent_pawns = self.opponent.pawns
        d = self.player.direction

        for i, j in self.player.pawns:
            if (i + d, j) not in opponent_pawns:
                moves.append(((i, j), (i + d, j)))

            if (i + d, j + 1) in opponent_pawns:
                moves.append(((i, j), (i + d, j + 1)))
```

```
            if (i + d, j - 1) in opponent_pawns:
                moves.append(((i, j), (i + d, j - 1)))

        return list(map(self.to_string, [(i, j) for i, j in moves]))

    # Define how to make a move
    def make_move(self, move):
        move = list(map(self.to_tuple, move.split(' ')))
        ind = self.player.pawns.index(move[0])
        self.player.pawns[ind] = move[1]

        if move[1] in self.opponent.pawns:
            self.opponent.pawns.remove(move[1])

    # Define what a loss looks like
    def loss_condition(self):
        return (any([i == self.opponent.goal_line
                for i, j in self.opponent.pawns])
                or (self.possible_moves() == []) )

    # Check if the game is over
    def is_over(self):
        return self.loss_condition()

    # Show the current status
    def show(self):
        f = lambda x: '1' if x in self.players[0].pawns else (
                '2' if x in self.players[1].pawns else '.')

        print("\n".join([" ".join([f((i, j))
                for j in range(self.size[1])])
                for i in range(self.size[0])]))

if __name__=='__main__':
    # Compute the score
    scoring = lambda game: -100 if game.loss_condition() else 0

    # Define the algorithm
    algorithm = Negamax(12, scoring)

    # Start the game
    game = GameController([AI_Player(algorithm),
            AI_Player(algorithm)])
    game.play()
    print('\nPlayer', game.nopponent, 'wins after', game.nmove , 'turns')
```

```
1 1 1 1
. . . .
. . . .
2 2 2 2

Move #1: player 1 plays A1 B1 :
. 1 1 1
1 . . .
. . . .
2 2 2 2

Move #2: player 2 plays D1 C1 :
. 1 1 1
1 . . .
2 . . .
. 2 2 2

Move #3: player 1 plays A2 B2 :
. . 1 1
1 1 . .
2 . . .
. 2 2 2

Move #4: player 2 plays D2 C2 :
. . 1 1
1 1 . .
2 2 . .
. . 2 2

Move #5: player 1 plays B1 C2 :
. . 1 1
. 1 . .
2 1 . .
. . 2 2

Move #6: player 2 plays C1 B1 :
. . 1 1
2 1 . .
. 1 . .
. . 2 2

Move #7: player 1 plays C2 D2 :
. . 1 1
2 1 . .
. . . .
. 1 2 2

Player 1 wins after 8 turns
```

Le bloc de code précédent implémente une variante du jeu Hexapawn, un jeu de stratégie basé sur des pions se déplaçant sur une grille, en utilisant la bibliothèque easyAI pour intégrer une intelligence artificielle.

Le plateau de jeu est une grille de 4x4 cases, où chaque joueur commence avec des pions alignés en rangée. Les pions avancent dans une seule direction et peuvent capturer en diagonale. Un joueur perd s'il n’a plus de coups possibles ou si l'adversaire atteint sa ligne de départ.

Le programme définit les déplacements possibles, effectue la mise à jour du plateau après chaque coup, et vérifie les conditions de victoire. Il affiche l'état du jeu après chaque mouvement sous forme de grille avec 1 pour un joueur et 2 pour l'autre.

L'IA utilise l'algorithme Negamax, avec une profondeur de recherche de 12 coups, pour analyser les meilleures stratégies. La partie se joue entre deux IA, et le programme annonce le gagnant ainsi que le nombre total de tours joués.

Ce code illustre une intelligence artificielle avancée appliquée aux jeux de stratégie, rendant le jeu autonome et compétitif.