```
import numpy as np
import nashpy as nash
from fractions import Fraction
import numpy as np
from scipy.optimize import linprog
SAVE = {}

GOOD = [[0, 9],
[1, 5],
[1, 8],
[1, 12],
[1, 17],
[2, 14],
[3, 4],
[3, 6],
[3, 7],
[3, 8],
[4, 5],
[4, 7],
[4, 8],
[5, 17],
[6, 12],
[6, 14],
[7, 10],
[7, 16],
[8, 17],
[10, 17],
[14, 15]]
def float_to_fraction(probability):
    return Fraction(probability).limit_denominator()
def BATTLE(G1,G2):
    stat1=[]
    stat2=[]

    for s in range(19):
        stat1.append(types[G1[0]][s]*types[G1[1]][s])
    for s in range(19):
        stat2.append(types[G2[0]][s]*types[G2[1]][s])
    if max(stat1[G2[0]],stat1[G2[1]]) < max(stat2[G1[0]],stat2[G1[1]]):
        return 1
    if max(stat1[G2[0]],stat1[G2[1]]) == max(stat2[G1[0]],stat2[G1[1]]):
        return 0
    if max(stat1[G2[0]],stat1[G2[1]]) > max(stat2[G1[0]],stat2[G1[1]]):
        return -1
```

```
NOR=[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 2, 1, 1, 1, 0]
#   [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, A, B, C, D, E, F, G, H]
WAT=[ 1,.5, 1, 2,.5, 1, 1, 2, 1, 1,.5, 1,.5, 1, 1, 1, 1, 1, 0]
PSY=[ 1, 1,.5, 1, 1, 1, 1, 1, 1, 2, 1, 2, 1, 1,.5, 2, 1, 1, 0]
#   [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, A, B, C, D, E, F, G, H]
ELE=[ 1, 1, 1,.5,.5, 1, 1, 1,.5, 1, 1, 1, 1, 1, 1, 1, 1, 2, 0]
STE=[.5, 1,.5, 1,.5,.5,.5,.5,.5, 1, 2,.5,.5, 0, 2, 1,.5, 2, 0]
#   [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, A, B, C, D, E, F, G, H]
DRA=[ 1,.5, 1,.5, 1, 2, 2,.5, 1, 1,.5, 1, 2, 1, 1, 1, 1, 1, 0]
FAI=[ 1, 1, 1, 1, 2, 0, 1, 1, 1, 1, 1,.5, 1, 2,.5,.5, 1, 1, 0]
#   [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, A, B, C, D, E, F, G, H]
GRA=[ 1,.5, 1,.5, 1, 1, 1,.5, 2, 1, 2, 2, 2, 2, 1, 1, 1,.5, 0]
FLY=[ 1, 1, 1, 2, 1, 1, 1,.5, 1, 1, 1,.5, 2, 1,.5, 1, 2, 0, 0]
#   [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, A, B, C, D, E, F, G, H]
GHO=[ 0, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1,.5, 1,.5, 0, 2, 1, 1, 0]
FIR=[ 1, 2, 1, 1,.5, 1,.5,.5, 1, 1,.5,.5,.5, 1, 1, 1, 2, 2, 0]
#   [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, A, B, C, D, E, F, G, H]
BUG=[ 1, 1, 1, 1, 1, 1, 1,.5, 2, 1, 2, 1, 1, 1,.5, 1, 2,.5, 0]
ICE=[ 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 2, 1,.5, 1, 2, 1, 2, 1, 0]
#   [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, A, B, C, D, E, F, G, H]
POI=[ 1, 1, 2, 1, 1, 1,.5,.5, 1, 1, 1,.5, 1,.5,.5, 1, 1, 2, 0]
FIG=[ 1, 1, 2, 1, 1, 1, 2, 1, 2, 1, 1,.5, 1, 1, 1,.5,.5, 1, 0]
#   [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, A, B, C, D, E, F, G, H]
DAR=[ 1, 1, 0, 1, 1, 1, 2, 1, 1,.5, 1, 2, 1, 1, 2,.5, 1, 1, 0]
ROC=[.5, 2, 1, 1, 2, 1, 1, 2,.5, 1,.5, 1, 1,.5, 2, 1, 1, 2, 0]
#   [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, A, B, C, D, E, F, G, H]
GRO=[ 1, 2, 1, 0, 1, 1, 1, 2, 1, 1, 1, 1, 2,.5, 1, 1,.5, 1, 0]
NON=[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0]
types=[NOR,WAT,PSY,ELE,STE,DRA,FAI,GRA,FLY,GHO,FIR,BUG,ICE,POI,FIG,DAR,ROC,GR
O,NON]
typenames=["NOR","WAT","PSY","ELE","STE","DRA","FAI","GRA","FLY","GHO","FIR","BUG","I
CE","POI","FIG","DAR","ROC","GRO","NON"]
def simulate(W,D,L):
    WR = W
    #DR = 0
    #DR = 4949/6863
    DR = D
    LR = L

    LS = [LR, DR, WR]
```

```
if DR >= 0:
    X = [[0,0,"SpecialW"]]
else:
    X = [[0,0,"SpecialL"]]
F=0

#for x in X:
#    A.append([x[0],x[1]])
for i in range(17+F):
    for j in range(17+F-i):
        I=i
        J=j+i+1
        X.append([I,J,"L"])
        X.append([I,J,"W"])
X1 = []
for x in X:
    nx = []
    for y in X:
        if x[2]=="SpecialL":
            if y[2]=="SpecialL" or y[2]=="SpecialW":
                nx.append(0)
            if y[2]=="L":
                nx.append(1)
            if y[2]=="W":
                nx.append(-1)
        if x[2]=="SpecialW":
            if y[2]=="SpecialL" or y[2]=="SpecialW":
                nx.append(0)
            if y[2]=="L":
                nx.append(-1)
            if y[2]=="W":
                nx.append(1)
        if x[2]=="L":
            if y[2]=="SpecialL":
                nx.append(-1)
            if y[2]=="SpecialW":
                nx.append(1)
            if y[2]=="L":
                nx.append(0)
            if y[2]=="W":
                nx.append(-LS[BATTLE(y,x)+1])
        if x[2]=="W":
            if y[2]=="SpecialL":
                nx.append(1)
```

```
            if y[2]=="SpecialW":
                nx.append(-1)
            if y[2]=="L":
                nx.append(LS[BATTLE(x,y)+1])
            if y[2]=="W":
                nx.append(0)
        X1.append(nx)


    def find_mixed_strategy_nash_equilibrium(game_matrix):
        # The number of rows and columns in the game matrix
        num_rows, num_cols = game_matrix.shape

        # Formulating the optimization problem for player 1 (maximize u)
        c = np.ones(num_rows) * -1  # Coefficients for the objective function (to maximize utility)
        A_eq = np.ones((1, num_rows))  # Equality constraints for the probabilities to sum to 1
        b_eq = np.array([1])  # The equality constraint value

        # Inequality constraints for the probabilities to be non-negative
        A_ub = -game_matrix.T  # Transposed game matrix with the signs flipped
        b_ub = np.zeros(num_cols)  # Zero vector for the inequality constraint values

        # Solve the linear programming problem for player 1
        result_player1 = linprog(c, A_ub=A_ub, b_ub=b_ub, A_eq=A_eq, b_eq=b_eq,
method='highs')

        # Extract the optimal mixed strategy probabilities for player 1
        mixed_strategy_player1 = result_player1.x

        # Formulating the optimization problem for player 2 (minimize v)
        c = np.ones(num_cols)  # Coefficients for the objective function (to minimize opponent's
utility)
        A_eq = np.ones((1, num_cols))  # Equality constraints for the probabilities to sum to 1
        b_eq = np.array([1])  # The equality constraint value

        # Inequality constraints for the probabilities to be non-negative
        A_ub = game_matrix  # Original game matrix
        b_ub = np.zeros(num_rows)  # Zero vector for the inequality constraint values

        # Solve the linear programming problem for player 2
        result_player2 = linprog(c, A_ub=A_ub, b_ub=b_ub, A_eq=A_eq, b_eq=b_eq,
method='highs')

        # Extract the optimal mixed strategy probabilities for player 2
```

```
        mixed_strategy_player2 = result_player2.x

        # Calculate the value of the game (both players should get the same value in zero-sum
games)
        game_value = 1 / result_player1.fun

        return mixed_strategy_player1, mixed_strategy_player2, game_value

    # Example usage:
    # Replace this example game matrix with the one you want to analyze
    game_matrix = np.array(X1)

    mixed_strategy_player1, mixed_strategy_player2, game_value =
find_mixed_strategy_nash_equilibrium(game_matrix)
    E = []
    E2 = []
    print()
    for i in range(len(mixed_strategy_player1)):
        if not float_to_fraction(mixed_strategy_player1[i]) == 0 and X[i][2]=="W":
            print(typenames[X[i][0]] + typenames[X[i][1]] + " - " +
str(float_to_fraction(2/(1-mixed_strategy_player1[0])*mixed_strategy_player1[i])))
            E.append([X[i][0],X[i][1],mixed_strategy_player1[i]])
    for i in range(len(mixed_strategy_player1)):
        if not float_to_fraction(mixed_strategy_player1[i]) == 0 and X[i][2]=="L":
            #print(typenames[X[i][0]] + typenames[X[i][1]] + " - " +
str(float_to_fraction(2/(1-mixed_strategy_player1[0])*mixed_strategy_player1[i])))
            E2.append([X[i][0],X[i][1],mixed_strategy_player1[i]])

    #print("Player 1's mixed strategy probabilities:", mixed_strategy_player1)
    #print("Player 2's mixed strategy probabilities:", mixed_strategy_player2)
    if X[0][2]=="SpecialW":
        print("Game value:",
float_to_fraction(2*mixed_strategy_player1[0]/(1-mixed_strategy_player1[0])))
    else:
        print("Game value (both players get the same):",
-float_to_fraction(mixed_strategy_player1[0]))
    #print(E)
    #print(E2)
    return float_to_fraction(2*mixed_strategy_player1[0]/(1-mixed_strategy_player1[0]))
def Score(RoundLeft,Bias):
    if Bias>RoundLeft:
        return 1
    if -Bias>RoundLeft:
        return 0
```

```
    if RoundLeft==0:
        return .5
    else:
        try:
            return SAVE[str(RoundLeft) + " - " + str(Bias)]
        except:
            SAVE[str(RoundLeft) + " - " + str(Bias)] =
simulate(Score(RoundLeft-1,Bias+1),Score(RoundLeft-1,Bias),Score(RoundLeft-1,Bias-1))
            print(str(RoundLeft) + " - " + str(Bias))
            return SAVE[str(RoundLeft) + " - " + str(Bias)]
for i in range(1):
    simulate(1,.5,0)
input
i=0
while True:
    Score(i,0)
    i+=1
    input()
```