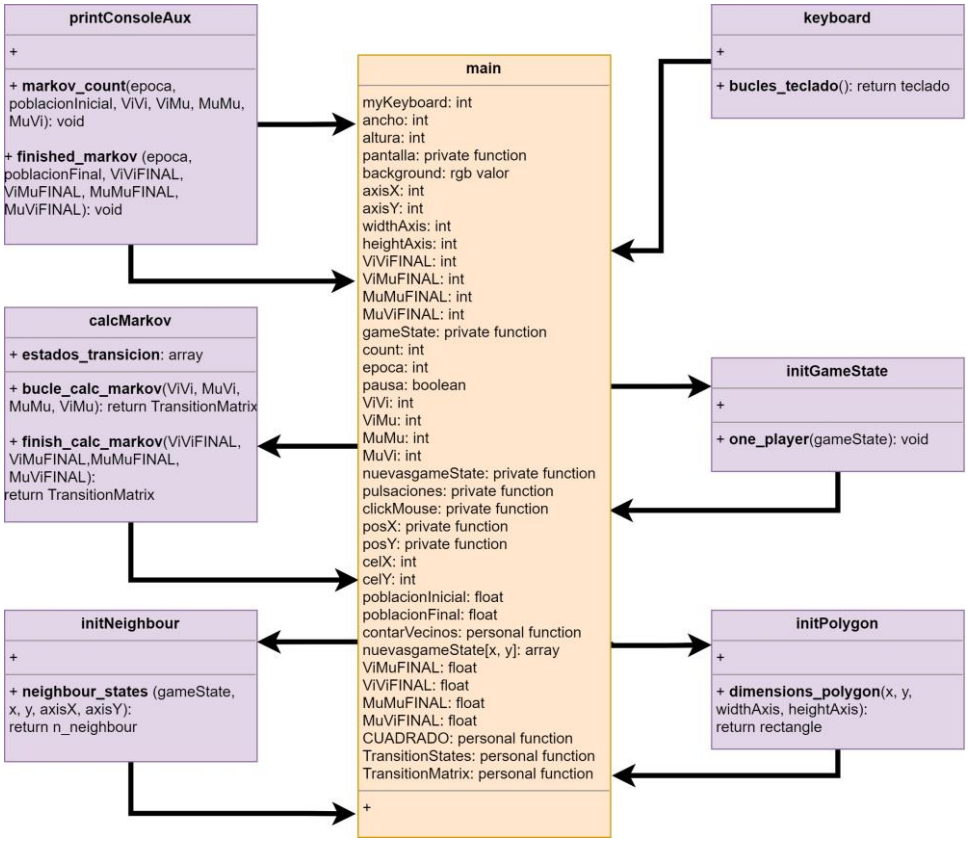
GAME OF LIFE – MARKOV CHAINS: Programación desarrollada en Python.

Ángel Manuel Correa Rivera. E-mail: angel.correa@usantoto.edu.co. Cod: 2279237. Facultad de Ingeniería de Sistemas. USTA.
Hugo Emmanuel Hernández Ramirez. E-mail: hugoe.hernandezr@usantoto.edu.co. Cod: 2345382. Facultad de Ingeniería de Sistemas. USTA.
Luis Felipe Narvaez Gomez. E-mail: luis.narvaez@usantoto.edu.co. Cod: 2312660. Facultad de Ingeniería de Sistemas. USTA.

Clase principal: [main.py]

El siguiente documento denota la realización de "el juego de la vida de Conway" basado en python con las cadenas de Markov impuestas en él, sabiendo que dichas cadenas son un proceso estocástico discreto en el que la probabilidad de que ocurra un evento depende solamente del evento inmediatamente anterior, proceso el cual se logró aplicar dentro del juego. Así las cosas, el código fue subdividido en clases y funciones separadas para dar un mayor índice de calidad y que este no quedase pesado a la lectura y entendimiento del usuario, quedando de la siguiente manera.




```python
# Librerias Generales
import pygame
import numpy as np
import time
import matplotlib.pyplot as plt
# Librerias Especificas personalizadas
import keyboard as key
import initGameState as gm
import initNeighbour as nb
import initPolygon as poly
import printConsoleAux as cli
```

```
import calcMarkov as calc

#Numero de Bucles:
myKeyboard = key.bucles_teclado()
print("El numero de bucles es de : ", myKeyboard)

pygame.init() #Inicializamos el juego
ancho, altura = 400, 400 #1000 Asignamos los pixeles a la pantalla.
pantalla = pygame.display.set_mode((altura, ancho)) #Creamos la pantalla
background = 25, 25, 25 #background de pantalla, con los colores de intensidad.
pantalla.fill(background) #Rellenamos la pantalla
axisX, axisY = 55, 55 #Determinar cuantas celdas se quieren en los 2 ejes
widthAxis = ancho / axisX #Ancho de las celdas
heightAxis = altura / axisY #Altura de las celdas

ViViFINAL = 0
ViMuFINAL = 0
MuMuFINAL = 0
MuViFINAL = 0

#zeros(dimensiones) : Crea y devuelve una referencia a un array cuyos elementos son
todos ceros.
gameState = np.zeros((axisX, axisY)) #Guardar todos los estados de las diferentes
celdas. 1 = viva, 0 = muerta
gm.one_player(gameState)

#control de la ejecucion en el juego
pausa = False

count = 0
epoca = 0
#zeros(dimensiones) : Crea y devuelve una referencia a un array cuyos elementos son
todos ceros.
while count < myKeyboard: #Bucles que recorren cada una de las celdas
    epoca = epoca + 1
    count = count + 1
    ViVi = 0
    ViMu = 0
    MuMu = 0
    MuVi = 0

    nuevasgameState = np.copy(gameState) #Evitamos que los cambios se hagan de forma
secuencial, y hace que sean todos de una.
    pantalla.fill(background)
    time.sleep(0.2)

    pulsaciones =  pygame.event.get() #Registro de pulsaciones en el teclado o mouse

    for event in pulsaciones:
        #Deteccion de pulsacion de cualquier tecla de teclado
```

```
        if event.type ==pygame.KEYDOWN:
            #cambio de la variable pausa si se oprime una tecla
            pausa = not pausa

    for y in range(0, axisX):
        for x in range(0, axisY):

            #El if da la pausa en caso de ser pulsada alguna tecla

            if not pausa:
                poblacionInicial = ViVi
                poblacionFinal = ViViFINAL + MuViFINAL

                contarVecinos = nb.neighbour_states(gameState, x, y, axisX, axisY)

                #Rule 1: Any live cell with fewer than two live neighbours dies, as if
by underpopulation.
                if gameState [x,y] == 1 and (contarVecinos < 2 or contarVecinos > 3):
                    nuevasgameState[x, y] = 0
                    ViMu = ViMu + 1
                    ViMuFINAL = ViMuFINAL + 1
                #Rule 2: Any live cell with two or three live neighbours lives on to the
next generation.
                elif gameState [x,y] == 1 and (contarVecinos == 2 or contarVecinos ==
3):
                    nuevasgameState[x,y] = 1
                    ViVi = ViVi + 1
                    ViViFINAL = ViViFINAL + 1
                #Rule 3: Any live cell with more than three live neighbours dies, as if
by overpopulation.
                #elif gameState[x, y] == 1 and (contarVecinos > 3):
                #   nuevasgameState[x, y] = 0
                #  ViMu = ViMu + 1
                #Rule 4: Any dead cell with exactly three live neighbours becomes a live
cell, as if by reproduction.
                elif gameState[x, y] == 0 and contarVecinos ==3:
                    nuevasgameState[x, y] = 1
                    MuVi = MuVi + 1
                    MuViFINAL = MuViFINAL + 1
                else:
                    MuMu = MuMu + 1
                    MuMuFINAL = MuMuFINAL + 1

                # Los puntos que defienen al poligono
                CUADRADO =  poly.dimensions_polygon(x, y, widthAxis, heightAxis)

                if nuevasgameState[x, y] == 0:
                    pygame.draw.polygon(pantalla,(128, 128, 128), CUADRADO, 1) #Dibujar
la pantalla para celula muerta
                else:
```

```
                    pygame.draw.polygon(pantalla, (180, 90, 110), CUADRADO, 0)  #
Dibujar la pantalla para celula viva

 #El if da la pausa en caso de ser pulsada alguna tecla
    if not pausa:
        cli.markov_count(epoca, poblacionInicial, ViVi, ViMu, MuMu, MuVi)
        TransitionStates = calc.estados_transicion
        TransitionMatrix = calc.bucle_calc_markov(ViVi, MuVi, MuMu, ViMu)
        print(TransitionMatrix)

        gameState = np.copy(nuevasgameState)  # Nuestro estado es el nuevo estado
        pygame.display.flip()

cli.finished_markov(epoca, poblacionFinal, ViViFINAL, ViMuFINAL, MuMuFINAL, MuViFINAL)
TransitionStates = calc.estados_transicion
TransitionMatrix = calc.finish_calc_markov(ViViFINAL,ViMuFINAL,MuMuFINAL,MuViFINAL)
print(TransitionMatrix)
```

CLASE SECUNDARIA: [keyboard.py]

```
#Numero de bucles
def bucles_teclado():
   while True:
      try:
         teclado = int( input( "Ingrese por favor el numero de bucles que desea: " ) )
         break
      except ValueError:
         print("El dato ingresado no es un numero, intentalo nuevamente ... ")

   return teclado
```

Esta Clase secundaria está preparada para recibir por parte del usuario el número de bucles que requiere para el simulador del juego de la vida de Conway. El dato ingresado está protegido por “Try/Cath”, puesto que en caso de que se ingrese un valor distinto a un número entero se informara del error y se pedirá el ingreso del dato.
Esta función es llamada en la clase principal por medio de importación de archivos locales e invocación igual a la siguiente:

```
import keyboard as key

#Numero de Bucles:
myKeyboard = key.bucles_teclado()
print("El numero de bucles es de : ", myKeyboard)
```

CLASE SECUNDARIA: [initGameState.py]

```python
import numpy as np

# Celdas iniciales con las que empezar el "juego"
def one_player(gameState):
    gameState[6,4] = 1
    gameState[6,5] = 1
    gameState[6,6] = 1
    gameState[5,6] = 1
    gameState[4,5] = 1
```

Esta función se encarga de inicializar unas celdas o células vivas (con valor 1) entre la matriz inicial del Juego de la vida constituida por celdas muertas (con valor 0). Para poder cambiar el estado de las células de la matriz inicial es necesario que esta última sea recibida por parámetro en la función y a continuación se invoque la celda o celdas especificas a cambiar su valor con sus coordenadas en (X, Y).

El parámetro de la matriz de ceros es enviado desde la case principal (main.py) y a la vez es invocado el resultado de las células inicializadas tal y como se muestra a continuación en el siguiente código:

```python
import initGameState as gm

ancho, altura = 800, 800
axisX, axisY = 25, 25

gameState = np.zeros((axisX, axisY))

gm.one_player(gameState)
```

Como podemos ver en el anterior código, primero se establecen el ancho y alto en pixeles de las cedas del tablero por cada uno de sus ejes (X, Y), luego conforme al tamaño total de la matriz (800 x 800), se rellena un tablero constituido de ceros “0”, esto con ayuda de “np.zeros” el cual crea y devuelve un arreglo cuyos valores son todos ceros. De esta matriz inicializada de (800 x 800) donde todos sus valores son ceros, se almacena en la variable “gameState” y es pasada por parámetro a nuestra función de la clase secundaria **initGameState**.

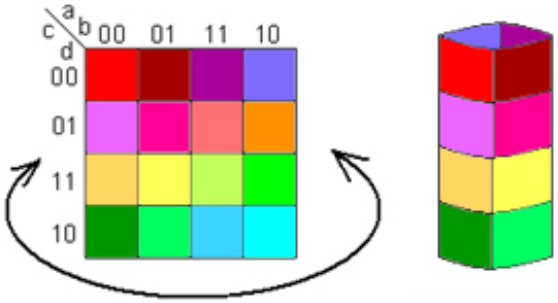
CLASE SECUNDARIA: [initNeighbour.py]

```python
def neighbour_states (gameState, x, y, axisX, axisY):
    ## % axisX, %axisY Se accede al tamaño del numero de la celda.
    n_neighbour =   gameState[(x-1) % axisX , (y-1) % axisY] + \
                    gameState[(x)   % axisX , (y-1) % axisY] + \
                    gameState[(x+1) % axisX , (y-1) % axisY] + \
                    gameState[(x-1) % axisX , (y)   % axisY] + \
                    gameState[(x+1) % axisX , (y)   % axisY] + \
                    gameState[(x-1) % axisX , (y+1) % axisY] + \
                    gameState[(x)   % axisX , (y+1) % axisY] + \
                    gameState[(x+1) % axisX , (y+1) % axisY]

    return n_neighbour
```

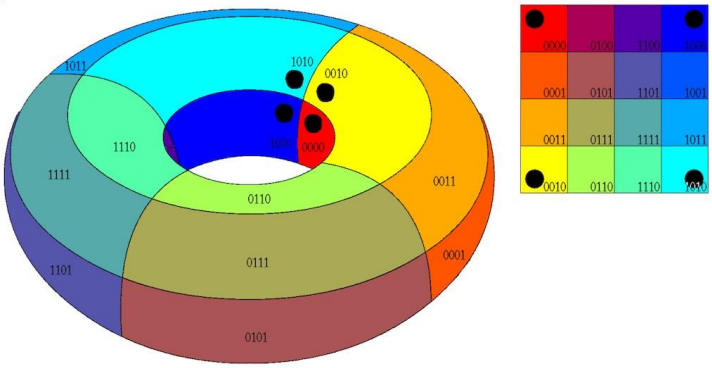
El cálculo de los vecinos a la celda está dado por la proximidad directa que tierne la celda central con todas aquellas celdas que lo circundan en vertical, horizontal, diagonal descendente y diagonal ascendente. Esto quiere decir que para cualquier celda central que tengamos para analizar, siempre tendremos ocho (8) vecinos rodeándolo en todo momento. Para entender esto mejor, supongamos que tiene una malla de 3x3 de dimensión, las coordenadas de cada celda de esta matriz son:

```
"""
            Calculo de vecinos cercanos

                Vecinos = Neighbour

[][][]          [(x-1 , y-1)] [(x , y-1)] [(x+1 , y-1)]
[][][]    =>    [(x-1 , y)  ] [(x , y)  ] [(x+1 , y)  ]
[][][]          [(x-1 , y+1)] [(x , y+1)] [(x+1 , y+1)]
"""
```

Como ya hemos mencionado antes, siempre en todo momento cada celda o célula central a analizar estará rodeada por sus 8 vecinos, sin embargo, esto es un poco complicado de entender cuando observamos celdas consecuentes al perímetro del tablero donde establecemos el campo de juego y es que, en ese lugar, cada celda no estará rodeada del mismo número de celdas vecinas que las que se encuentran al interior del perímetro del tablero.
Para hacer que esto se cumpla podemos ayudarnos del concepto de la algebra Booleana y los mapas de Karnaugh, donde el operar tableros de matrices de compuertas lógicas como AND, OR, XOR, etc; se debe poder trazar una continuidad entre el valor de la casilla del perímetro de la derecha con la primera del perímetro de la izquierda, para esto imagine el tablero en una hoja de papel, cuyos bordes extremos en horizontal, realmente se tocan generando de un plano a un cilindro hueco.



Ahora bien, para que incluya las celdas de las esquinas de la tabla tengan todos sus vecinos, debemos juntar el extremo inferior y superior de la tabla, para esto vasta con tomar el cilindró que ya tenemos y juntar sus bocas lo que nos generaría un esquema de una figura toroidal como se muestra a continuación.



Para aplicar este concepto en el tablero de juego basta con multiplicar cada uno de los ejes de los vecinos por su respectivo valor inicial del eje como se observó en el código, de esta manera cada celda siempre estará rodeada de 8 vecinos y tendrá un comportamiento de caminata infinita por la matriz preparada.

Para poder invocar esta función en el main tenemos lo siguiente:

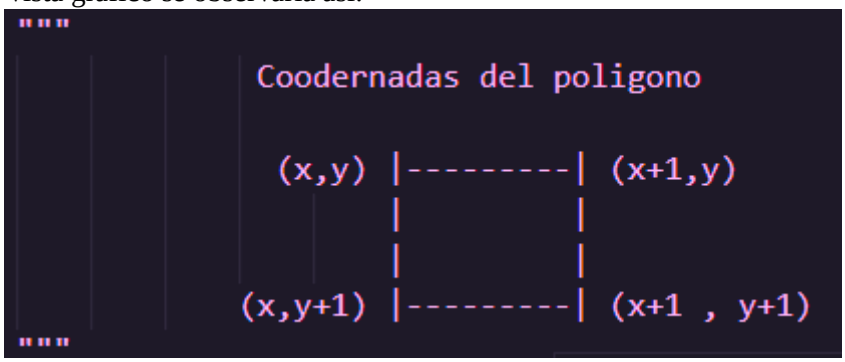
```
import numpy as np
import initNeighbour as nb

axisX, axisY = 55, 55
gameState = np.zeros((axisX, axisY))
while count < myKeyboard:
   for y in range(0, axisX):
     for x in range(0, axisY):
            contarVecinos = nb.neighbour_states(gameState, x, y, axisX, axisY)
```

Como puede ver en el código, la parte que se destina a los vecinos está dentro de dos ciclos “for”, uno por cada eje que se debe recorrer en la matriz o tablero de juego dada por la variable “gameState” con las dimensiones de “axisX” y “axisY”.

Clase secundaria: [initPolygon.py]

```
def dimensions_polygon(x, y, widthAxis, heightAxis):
    rectangle = [((x)   * widthAxis , y     * heightAxis),
                 ((x+1) * widthAxis , y     * heightAxis),
                 ((x+1) * widthAxis , (y+1) * heightAxis),
                 ((x)   * widthAxis , (y+1) * heightAxis)]

    return rectangle
```

La función de esta clase es sencilla, pues muestra los vértices del polígono del cual está formada la célula. El polígono que utilizamos es uno regular de forma cuadrada por lo que posee 4 vértices entre los cuales su separación entre vértice y vértice esta dado por la dimensión del eje en altura y anchura dada en pixeles; tal y como se muestra en el anterior código.
Visto desde el punto de vista grafico se observaría así:



Para que la función tenga uso necesita de los parámetros de ingreso dados desde la clase principal, esto se puede ver en el siguiente código.

```
import numpy as np
import initPolygon as poly

axisX, axisY = 55, 55 #Determinar cuántas celdas se quieren en los 2 ejes
widthAxis = ancho / axisX #Ancho de las celdas
heightAxis = altura / axisY #Altura de las celdas
```

```
while count < myKeyboard:
      for y in range(0, axisX):
             for x in range(0, axisY):
                # Los puntos que definen al polígono
                CUADRADO =  poly.dimensions_polygon(x, y, widthAxis, heightAxis)
```

Clase secundaria: [printConsoleAux.py]

```
def markov_count(epoca, poblacionInicial, ViVi, ViMu, MuMu, MuVi):
    print("=============================================================================")
    print("Epoca: ", epoca)
    print("La poblacion de la epoca ", " es de : ", poblacionInicial)
    print("La poblacion que permanecio viva es de : ", ViVi)
    print("La poblacion que murio fue de : ", ViMu)
    print("La poblacion que se matuvo muerta es de : ", MuMu)
    print("La poblacion que revivio con las esferas del dragon es de : ", MuVi)
    print("=============================================================================")


def finished_markov(epoca, poblacionFinal, ViViFINAL, ViMuFINAL, MuMuFINAL, MuViFINAL):
    print("=============================================================================")
    print("El total de epocas fueron: ", epoca)
    print("La poblacion de todas las generaciones fue de : ", poblacionFinal)
    print("La poblacion que permanecio viva en general fue de : ", ViViFINAL)
    print("La poblacion que paso de viva a muerta en general fue de  : ", ViMuFINAL)
    print("La poblacion que se mantuvo muerta en toda la simulacion fue de : ",
MuMuFINAL)
    print("La poblacion que revivio en general con las esferas del dragon es de : ",
MuViFINAL)
    print("=============================================================================")
```

Tal y como se pues observar en el código, la clase cuenta con dos funciones cuya labor no es más que mostrar los valores de las variables mediante el uso de la consola CLI. Si bien esto puede darse dentro de la misma clase principal, se destina otra secundaria con la finalidad que “main.py” no esté tan saturada de código. Las variables que utiliza para graficar en CLI llegan por parámetro y son mandadas dentro de la clase principal al momento de su invocación.

```
import printConsoleAux as cli

while count < myKeyboard:
   for y in range(0, axisX):
      for x in range(0, axisY):
         #Aquí realmente bajo las reglas del juego de la vida y el conteo de Markov
generan las variables que más tarde se envían a las funciones que necesitamos.
        cli.markov_count(epoca, poblacionInicial, ViVi, ViMu, MuMu, MuVi)

cli.finished_markov(epoca, poblacionFinal, ViViFINAL, ViMuFINAL, MuMuFINAL, MuViFINAL)
```

Clase secundaria: [calcMarkov.py]

```python
#Dos dimensiones, lista dentro otra lista [matriz], para las 4 posibles transiciones
estados_transicion = [["ViVi", "ViMu"], ["MuVi", "MuMu"]]

def bucle_calc_markov(ViVi, MuVi, MuMu, ViMu):
    TransitionMatrix = [[(ViVi / (ViVi + ViMu)),
                         (ViMu / (ViVi+ ViMu))],
                        [(MuVi / (MuVi + MuMu)),
                         (MuMu / (MuVi + MuMu))]]
    return TransitionMatrix

def finish_calc_markov(ViViFINAL,ViMuFINAL,MuMuFINAL,MuViFINAL):
    #Matriz de transicion
    TransitionMatrix = [[(ViViFINAL/(ViViFINAL+ViMuFINAL)),
                         (ViMuFINAL/(ViViFINAL+ViMuFINAL))],
                        [(MuViFINAL/(MuViFINAL+MuMuFINAL)),
                         (MuMuFINAL/(MuViFINAL+MuMuFINAL))]]
    return TransitionMatrix
```

Esta clase también sirve como segregación a una función externa del cálculo de Markov. Para funcionar cada una de las dos funciones recibe de la clase principal las variables por parámetro. En este caso tenemos dos funciones, la primera se encarga de ser el cálculo dado por cada uno de los ciclos dentro del bucle mientras que la segunda función agrupa los resultados de cada uno de los ciclos y hace la respectiva operación consecuente.

El funcionamiento de Markov para el juego e la vida es el siguiente:

1. Lo primero que hice fue crear 4 atributos de las 4 transiciones posibles fuera del while principal, estas sirven para ir acumulando los estados de las transiciones y poder sumarlo y sacar la probabilidad general al finalizar la simulación

```python
ViViFINAL = 0
ViMuFINAL = 0
MuMuFINAL = 0
MuViFINAL = 0
```

2. Después cree 4 atributos más dentro del while para poder almacenar época por época los estados.

```python
ViVi = 0
ViMu = 0
MuMu = 0
MuVi = 0
```

3. Dependiendo cada regla de las que contiene el juego de la vida, iba agregando a los 4 atributos

```
#Rule 1: Any live cell with fewer than two live neighbours dies, as if by underpopulation.
if celulas [x,y] == 1 and (contarVecinos < 2 or contarVecinos > 3):
    nuevasCelulas[x, y] = 0
    ViMu = ViMu + 1
    ViMuFINAL = ViMuFINAL + 1


#Rule 2: Any live cell with two or three live neighbours lives on to the next generation.
elif celulas [x,y] == 1 and (contarVecinos == 2 or contarVecinos == 3):
    nuevasCelulas[x,y] = 1
    ViVi = ViVi + 1
    ViViFINAL = ViViFINAL + 1


#Rule 3: Any live cell with more than three live neighbours dies, as if by overpopulation.
#elif celulas[x, y] == 1 and (contarVecinos > 3):
 #   nuevasCelulas[x, y] = 0
  #  ViMu = ViMu + 1

#Rule 4: Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.
elif celulas[x, y] == 0 and contarVecinos ==3:
    nuevasCelulas[x, y] = 1
    MuVi = MuVi + 1
    MuViFINAL = MuViFINAL + 1

else:
    MuMu = MuMu + 1
    MuMuFINAL = MuMuFINAL + 1
```

Por ejemplo, en la regla número 1, si la célula tiene menos de 2 celular o más de 3, esta pasaría de viva a muerta, lo cual significa, que le sumamos 1 a nuestro atributo ViMu (Vivo a Muerto), y también en ViMuFinal, para ir almacenando la época 1, 2, 3 sucesivamente.

4. Por último, creamos la matriz de estados y la matriz de transición, la cual contiene las fórmulas de:

```
print("═══════════════════════════════════════════════════════════════")
print("Epoca: ", epoca)
print("La poblacion de la epoca ", " es de : ", poblacionInicial)
print("La poblacion que permanecio viva es de : ", ViVi)
print("La poblacion que murio fue de : ", ViMu)
print("La poblacion que se matuvo muerta es de : ", MuMu)
print("La poblacion que revivio con las esferas del dragon es de : ", MuVi)
print("═══════════════════════════════════════════════════════════════")
TransitionStates = [["ViVi", "ViMu"], ["MuVi", "MuMu"]]
TransitionMatrix = [[(ViVi / (ViVi + ViMu)), (ViMu / (ViVi+ ViMu))],
                    [(MuVi / (MuVi + MuMu)), (MuMu / (MuVi + MuMu))]]
print(TransitionMatrix)
```