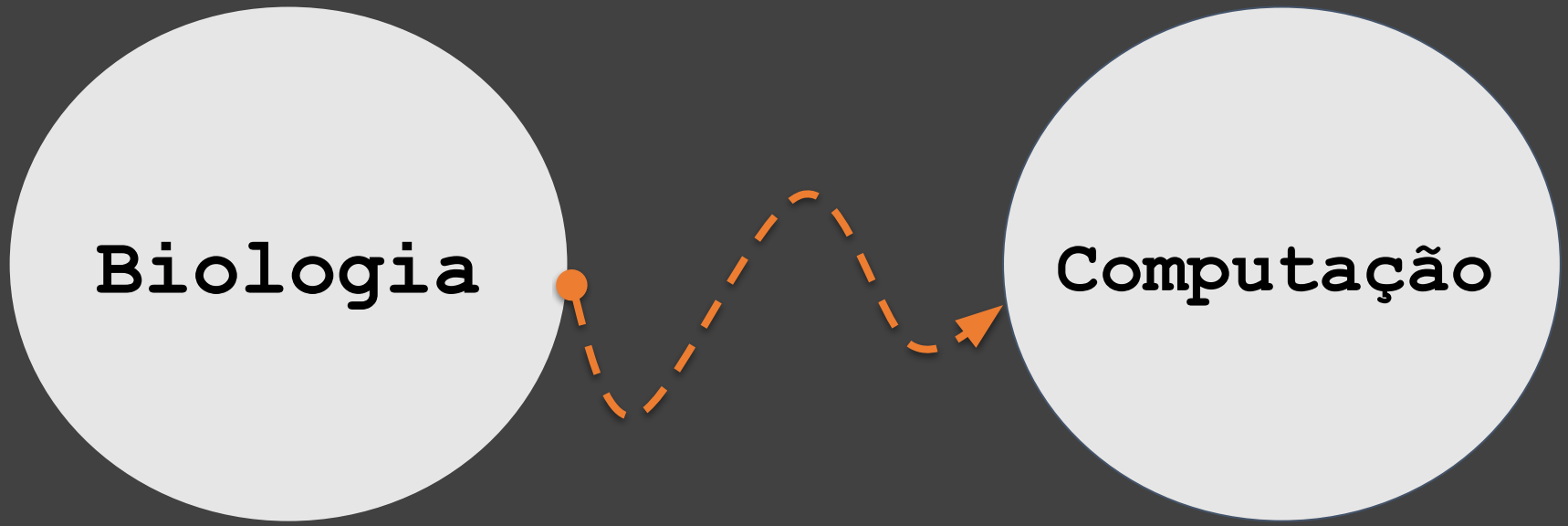
Workshop de Algoritmos Genéticos

Grupo Turing

Introdução



Evolução Biológica

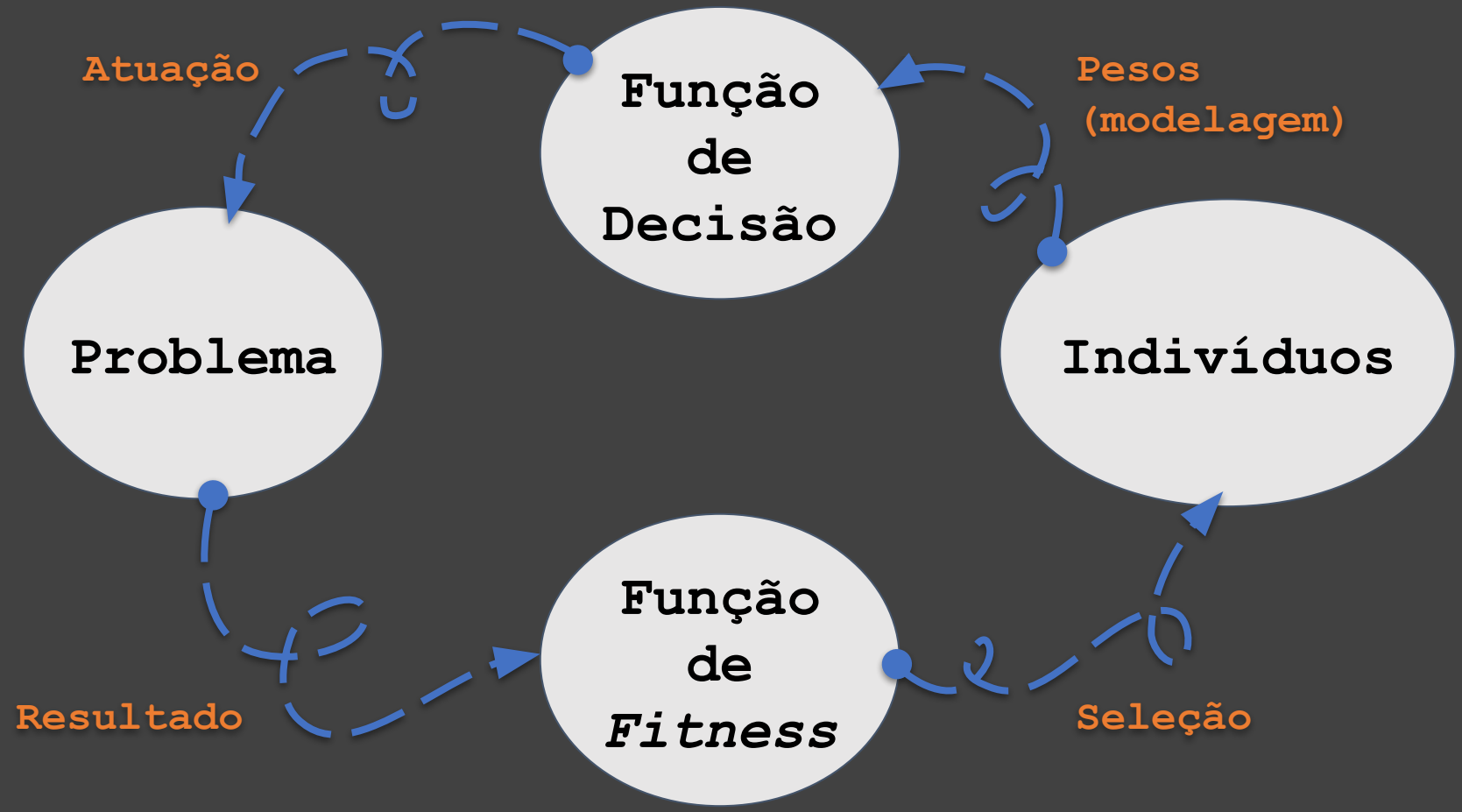
População de uma espécie em um determinado **meio**

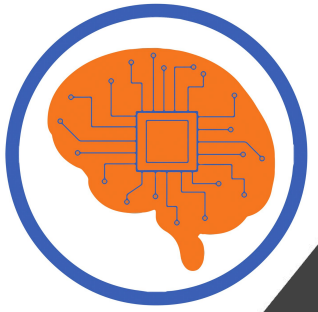
Seleção Natural

Os indivíduos melhor adaptados ao meio tendem a se reproduzir, enquanto os demais tendem a morrer.

A adaptação ou não de um indivíduo ao meio está primariamente relacionada às suas características determinadas por via genética.

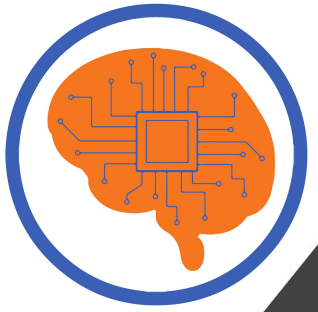
O tal do algoritmo Genético...





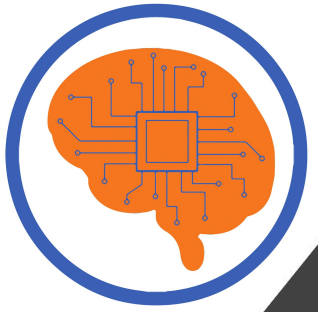
Elucidando termos...

- **Estado** → Lista de números que representam estado atual do jogo (por exemplo, a posição do dinossauro e dos obstáculos, …)
- **Indivíduo** → Lista com pesos
- **Peso** → Número que representa a importância de cada característica do estado
- **População** → Conjunto de indivíduos
- **Geração** → População de uma dada iteração do algoritmo
- **Função de Fitness** → Função de avaliação do indivíduo perante ao problema
- **Reprodução** → Processo em que os melhores indivíduos são combinados dando origem a uma nova geração
- **Critério de Parada** → Determina o fim do algoritmo, pode ser por tempo, nº de gerações ou valor satisfatório de fitness



Analogia computacional-biológica

Na Biologia...	... Na Computação
Cromossomo	Indivíduo
Gene	Peso (valor)
Lócus	Local do peso no vetor de pesos
Genótipo	Vetor de pesos que representa o indivíduo
Fenótipo	Interpretação/resultado do vetor na função de decisão



O Algoritmo genético

ENTRADA: População Inicial (aleatória)

Função de *Fitness*

Critério de Parada

REPITA (até que o critério de parada seja atendido):

PASSO 1: Aplicar a função de fitness a cada indivíduo

PASSO 2: Selecionar os x melhores indivíduos

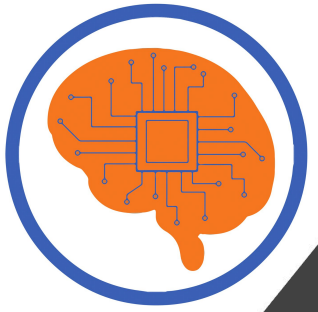
PASSO 3: Reprodução

- Aplicar o *crossover* a um par (com prob = p)
- Aplicar mutação (com prob = p')

PASSO 4: Formar uma nova população com os filhos gerados

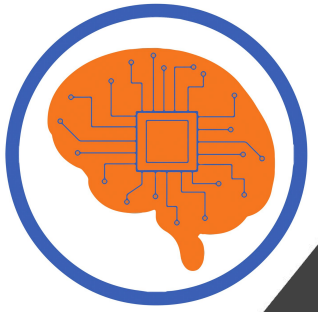
SAÍDA: Melhor indivíduo presente na geração final

A Proposta



O algoritmo do T-Rex

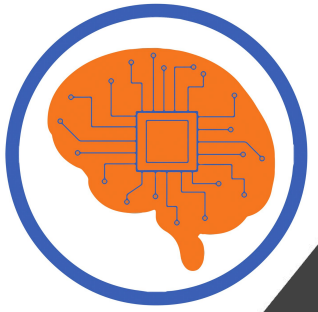
- O jogo tem um **estado**
- O indivíduo toma **ações**
- O **indivíduo** é uma **matriz**



O Estado

- É uma lista de números que explicam as características atuais do jogo
- Ex (um estado de tamanho 5):

$$S = \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \end{bmatrix} = \begin{bmatrix} 2.1 \\ 4.5 \\ 3.2 \\ 1.5 \\ 7.4 \end{bmatrix} \begin{matrix} \text{(velocidade)} \\ \text{(coordenada x do próximo obstáculo)} \\ \text{(coordenada y do próximo obstáculo)} \\ \text{(...)} \\ \text{(...)} \end{matrix}$$



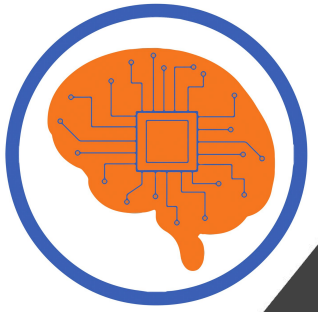
As ações

- 0, 1 e 2, correspondendo a pular, agachar e ir para a frente
- O **valor** de uma ação: indica o quão boa ela é
- O valor depende do estado

- Simplificação: função linear

$$v = a \cdot s_0 + b \cdot s_1 + c \cdot s_2 + d \cdot s_3 + e \cdot s_4$$

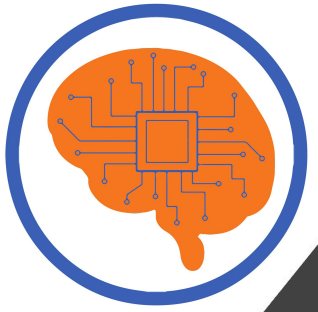
- Objetivo: achar a, b, c, d, e (os pesos)



O VALOR DAS AÇÕES

$$
\begin{aligned}
v_0 &= a \cdot s_0 + b \cdot s_1 + c \cdot s_2 + d \cdot s_3 + e \cdot s_4 \\
v_1 &= f \cdot s_0 + g \cdot s_1 + h \cdot s_2 + i \cdot s_3 + j \cdot s_4 \\
v_2 &= k \cdot s_0 + l \cdot s_1 + m \cdot s_2 + n \cdot s_3 + o \cdot s_4
\end{aligned}
$$

$$
\overbrace{\begin{pmatrix} a & b & c & d & e \\ f & g & h & i & j \\ k & l & m & n & o \end{pmatrix}}^{C}
\overbrace{\begin{pmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \end{pmatrix}}^{S}
=
\overbrace{\begin{pmatrix} v_0 \\ v_1 \\ v_2 \end{pmatrix}}^{V}
$$



O valor das ações

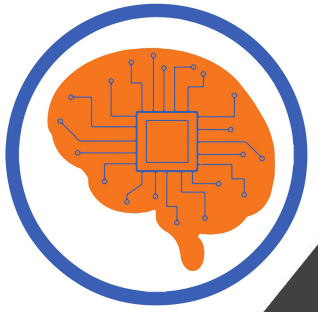
Ou seja,

$$V = CS$$

onde:

- **V** é um vetor com os valores das 3 ações
- **C** é uma matriz 3x5 com números reais
- **S** é um vetor com os valores do estado

Tendo **C**, basta calcular **V** para saber qual ação tomar em cada estado

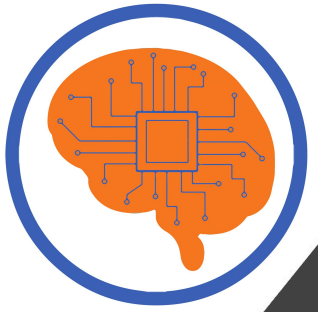


O valor das ações - Python

$$V = CS$$

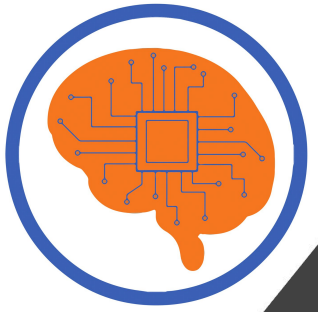
- Em python, multiplicação de matrizes é representada pelo símbolo “@”:

```
C = np.array([[1, 2, 3],
              [2, 1, 3],
              [3, 2, 1]])
S = [1, 2, 3]
V = C @ S
# V = [14, 13, 10]
```



Como achar a matriz "C"?

- Utilizando **algoritmos genéticos**
- Cada matriz **C** é um **indivíduo**
- **Mutação:** mudar um número da matriz
- **Crossover:** trocar números de uma matriz por números de outra matriz



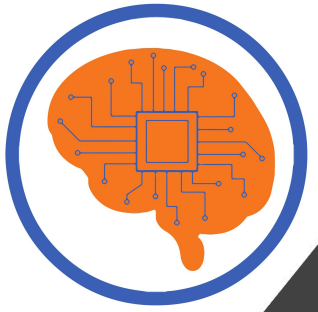
Mutação

$$\begin{pmatrix} 2.1 & 3.2 & 1.1 & 9.2 & 3.6 \\ 4.3 & 1.5 & 8.3 & 5.6 & 5.4 \\ 1.8 & 8.8 & 4.9 & 3.6 & 1.4 \end{pmatrix} \longrightarrow \begin{pmatrix} 2.1 & 3.2 & 1.1 & 9.2 & 3.6 \\ 4.3 & 2.4 & 8.3 & 5.6 & 5.4 \\ 1.8 & 8.8 & 4.9 & 9.1 & 1.4 \end{pmatrix}$$

Como trocar o número?

Para um número aleatório r:

- $a \mapsto ra$
- $a \mapsto a + r$
- $a \mapsto r$

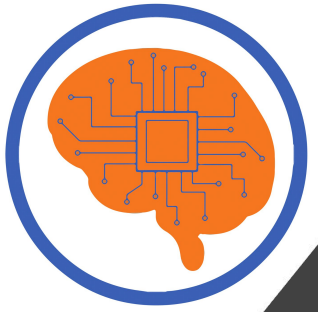


Crossover

$$\begin{pmatrix} 2.1 & 3.2 & 1.1 & 9.2 & 3.6 \\ 4.3 & 1.5 & 8.3 & 5.6 & 5.4 \\ 1.8 & 8.8 & 4.9 & 3.6 & 1.4 \end{pmatrix}$$

$$\begin{pmatrix} 4.5 & 8.9 & 1.4 & 4.4 & 9.8 \\ 1.2 & 2.3 & 3.4 & 4.5 & 5.6 \\ 6.7 & 7.8 & 8.9 & 9.1 & 1.2 \end{pmatrix}$$

$$\rightarrow \begin{pmatrix} 2.1 & 3.2 & 1.1 & 9.2 & 9.8 \\ 1.2 & 1.5 & 3.4 & 5.6 & 5.4 \\ 1.8 & 8.8 & 4.9 & 3.6 & 1.4 \end{pmatrix}$$

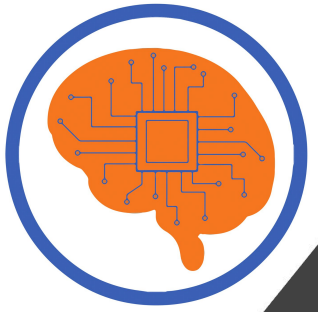


COMO LIDAR COM PROBABILIDADES?

Em alguns momentos, vamos querer que um evento ocorra com probabilidade p (mutação e crossing over).

Para fazer isso basta gerar um número aleatório p entre 0 e 1 (nossa variável aleatória com distribuição uniforme) e verificar se ele está dentro de um intervalo do tamanho da probabilidade que queremos. Exemplificando:

A probabilidade de um número aleatório entre 0 e 1 estar entre 0 e 0,2 é 20%

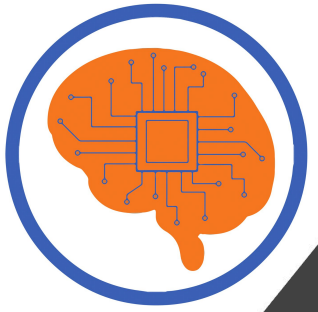


COMO LIDAR COM PROBABILIDADES?

A probabilidade de um número aleatório entre 0 e 1 estar entre 0,95 e 1 é 5%

Se quero que um evento ocorra com probabilidade 20%:

1. Gero um p aleatório entre 0 e 1
2. Verifico se ele está num intervalo de tamanho 0,2 (por exemplo entre 0,8 e 1)
3. Se ele estiver, executo o evento
4. Caso ele não esteja, não executo o evento

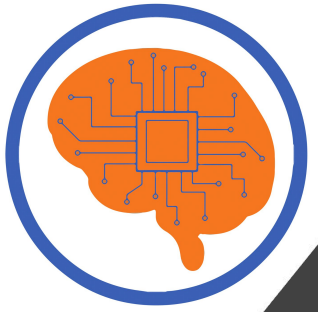


O que devemos fazer

Os arquivos:

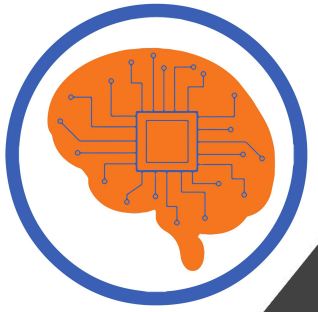
- **genetico.py** (treina o algoritmo)
- **chrome_trex.zip** (contém o código do jogo)

Desses arquivos, vocês trabalharão somente no **genetico.py**



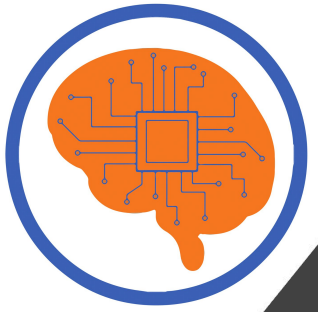
O que devemos fazer

```
# genetico.py

# As funções abaixo
def populacao_aleatoria(n):
def valor_das_acoes(individuo, estado):
def melhor_jogada(individuo, estado):
def mutacao(individuo):
def crossover(individuo1, individuo2):
def calcular_fitness(jogo, individuo):
def proxima_geracao(populacao, fitness):

# E o código principal, que roda o algoritmo
# usando as funções acima
```



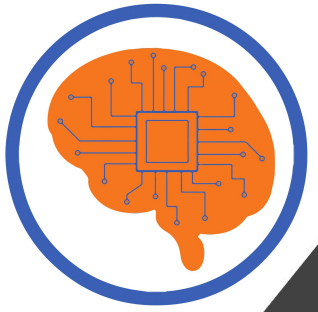
O que devemos fazer

```
# SETUP

import numpy as np
import random
import sys
sys.path.append('chrome_trex.zip')

from chrome_trex import DinoGame


# Sinta-se livre para brincar com os valores abaixo

CHANCE_MUT = .2       # Chance de mutação de um peso qualquer
CHANCE_CO = .25       # Chance de crossing over de um peso qualquer
NUM_INDIVIDUOS = 15   # Tamanho da população
NUM_MELHORES = 8      # Número de indivíduos que são mantidos
                      # de uma geração para a próxima
```



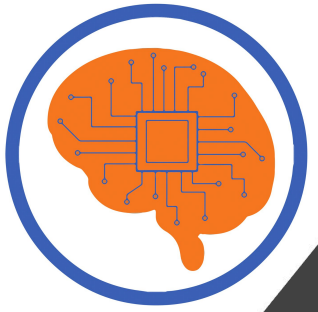
O QUE DEVEMOS FAZER

```
# FUNÇÕES FORNECIDAS

def ordenar_lista(lista, ordenacao, decrescente=True):
    """
    Argumentos da Função:
        lista: lista de números a ser ordenada.
        ordenacao: lista auxiliar de números que define a prioridade da
        ordenação.
        decrescente: variável booleana para definir se a lista `ordenacao`
        deve ser ordenada em ordem crescente ou decrescente.
    Saída:
        Uma lista com o conteúdo de `lista` ordenada com base em
        `ordenacao`.
    Por exemplo,
        ordenar_lista([2, 4, 5, 6], [7, 2, 5, 4])
        # retorna [2, 5, 6, 4]
        ordenar_lista([1, 5, 4, 3], [3, 8, 2, 1])
        # retorna [5, 1, 4, 3]
    """
```



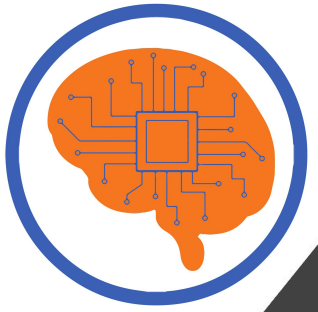
O QUE DEVEMOS FAZER

```
# FUNÇÕES FORNECIDAS

def mostrar_melhor_individuo(jogo, populacao, fitness):
    """
    Argumentos da Função:
        jogo: objeto que representa o jogo.
        populacao: lista de indivíduos.
        fitness: lista de fitness, uma para cada indivíduo.
    Saída:
        Não há saída. Simplesmente mostra o melhor indivíduo de uma
        população.
    """

    # VOCÊ NÃO PRECISA MEXER NESSA FUNÇÂO
```



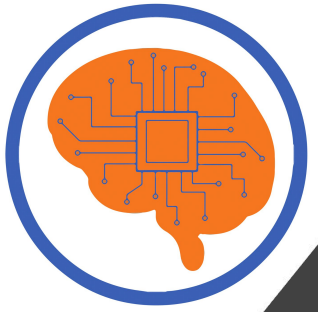
O QUE DEVEMOS FAZER

```
# FUNÇÕES QUE VOCÊS DEVEM IMPLEMENTAR

def populacao_aleatoria(n):
    """
    Argumentos da Função:
        n: Número de indivíduos
    Saída:
        Uma população aleatória. População é uma lista de indivíduos,
        e cada indivíduo é uma matriz 3x10 de pesos (números).
        Os indivíduos podem tomar 3 ações (0, 1, 2) e cada linha da matriz
        contém os pesos associados a uma das ações.
    """
    # Referência: np.random.uniform()
    #             list.append()
    #             for loop (for x in lista)
```



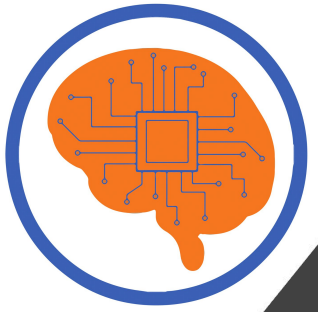
O QUE DEVEMOS FAZER

```
# FUNÇÕES QUE VOCÊS DEVEM IMPLEMENTAR

def valor_das_acoes(individuo, estado):
    """
    Argumentos da Função:
        individuo: matriz 3x10 com os pesos do indivíduo.
        estado: lista com 10 números que representam o estado do jogo.
    Saída:
        Uma lista com os valores das ações no estado `estado`. Calcula os
        valores das jogadas como combinações lineares dos valores do estado,
        ou seja, multiplica a matriz de pesos pelo estado.
    """
    # Referência: multiplicação de matrizes (A @ B)
```



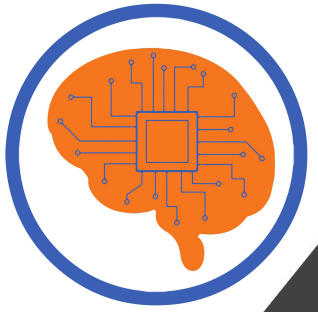
O QUE DEVEMOS FAZER

```
# FUNÇÕES QUE VOCÊS DEVEM IMPLEMENTAR

def melhor_jogada(individuo, estado):
    """
    Argumentos da Função:
        individuo: matriz 3x10 com os pesos do indivíduo.
        estado: lista com 10 números que representam o estado do jogo.
    Saída:
        A ação de maior valor (0, 1 ou 2) calculada pela função
        valor_das_acoes.
    """
    # Referência: np.argmax()
```



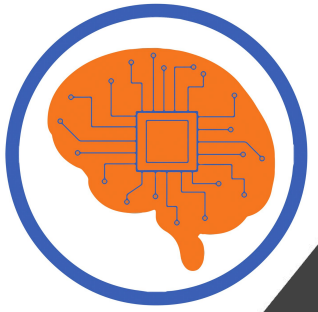
O QUE DEVEMOS FAZER

```
# FUNÇÕES QUE VOCÊS DEVEM IMPLEMENTAR

def mutacao(individuo):
    """
    Argumentos da Função:
        individuo: matriz 3x10 com os pesos do indivíduo.
    Saída:
        Essa função não tem saída. Ela apenas modifica os pesos do
        indivíduo, de acordo com chance CHANCE_MUT para cada peso.
    """
    # Referência: for loop (for x in lista)
    #             np.random.uniform()

    # A modificação dos pesos pode ser feita
    # de diversas formas (vide slides)
```



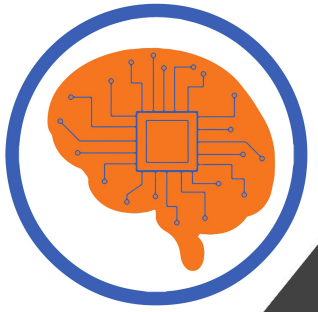
O QUE DEVEMOS FAZER

```
# FUNÇÕES QUE VOCÊS DEVEM IMPLEMENTAR

def crossover(individuo1, individuo2):
    """
    Argumentos da Função:
        individuoX: matriz 3x10 com os pesos do individuoX.
    Saída:
        Um novo indivíduo com pesos que podem vir do `individuo1`
        (com chance 1-CHANCE_CO) ou do `individuo2` (com chance CHANCE_CO),
        ou seja, é um cruzamento entre os dois indivíduos. Você também pode
        pensar que essa função cria uma cópia do `individuo1`, mas com
        chance CHANCE_CO, copia os respectivos pesos do `individuo2`.
    """
    # Referência: for loop (for x in lista)
    #             np.random.uniform()
```



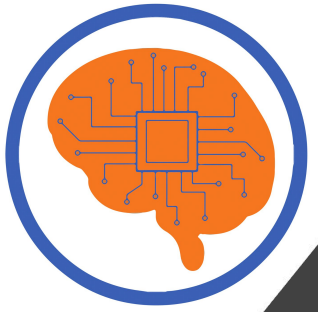
O QUE DEVEMOS FAZER

```
# FUNÇÕES QUE VOCÊS DEVEM IMPLEMENTAR

def calcular_fitness(jogo, individuo):
    """
    Argumentos da Função:
        jogo: objeto que representa o jogo.
        individuo: matriz 3x10 com os pesos do individuo.
    Saída:
        O fitness calculado de um indivíduo. Esse cálculo é feito simulando
        um jogo e calculando o fitness com base nessa simulação. O modo mais
        simples é usando fitness = score do jogo.
    """
    # Referência: while loop (while condition)
    #             np.random.uniform()
```



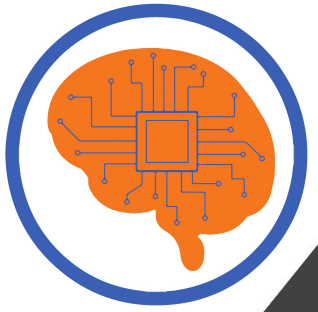
O QUE DEVEMOS FAZER

```
# FUNÇÕES QUE VOCÊS DEVEM IMPLEMENTAR

def calcular_fitness(jogo, individuo):
    # Você precisará da função melhor_jogada()
    # O jogo é simulado usando:
    #   jogo.reset()
    #   jogo.game_over
    #   jogo.step(acao)  # Toma a ação `acao` e vai para o próximo frame
    #   jogo.get_score()
    #   jogo.get_state()

    jogo.reset()
    while not jogo.game_over:
        acao = ?
        jogo.step(acao)
    return jogo.get_score()
```



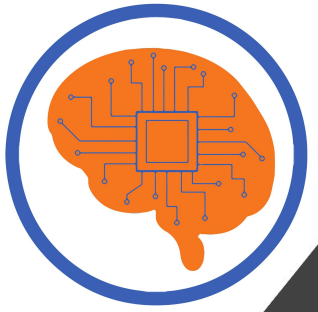
O QUE DEVEMOS FAZER

```
# FUNÇÕES QUE VOCÊS DEVEM IMPLEMENTAR

def proxima_geracao(populacao, fitness):
    """
    Argumentos da Função:
        populacao: lista de indivíduos.
        fitness: lista de fitness, uma para cada indivíduo.
    Saída:
        A próxima geração com base na população atual.
        Para criar a próxima geração, segue-se o seguinte algoritmo:
          1. Colocar os melhores indivíduos da geração atual na próxima
             geração.
          2. Até que a população esteja completa:
             2.1. Escolher aleatoriamente dois indivíduos da geração atual.
             2.2. Criar um novo indivíduo a partir desses dois indivíduos
                  usando crossing over.
             2.3. Mutar esse indivíduo.
             2.4. Adicionar esse indivíduo na próxima geração
    """
```



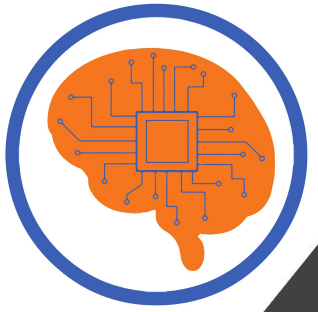
O QUE DEVEMOS FAZER

```
# FUNÇÕES QUE VOCÊS DEVEM IMPLEMENTAR

def proxima_geracao(populacao, fitness):
    # Referência: random.choices(populacao, weights=None, k=2)
    #             while loop (while condition)
    #             lista[a:b]

    # Dica: lembre-se da função `ordenar_lista(lista, ordenacao)`.
    # Você precisará várias funções que você já implementou.
```



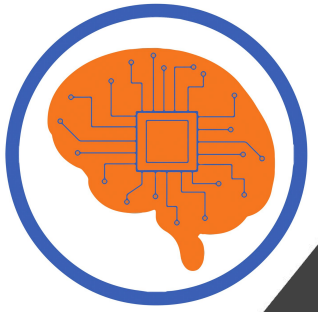
O QUE DEVEMOS FAZER

```
# FUNÇÕES QUE VOCÊS DEVEM IMPLEMENTAR

def proxima_geracao(populacao, fitness):
    proxima_ger = []

    # Adicionar os melhores indivíduos da geração atual na próxima geração

    while len(proxima_ger) < NUM_INDIVIDUOS:
        # Selecionar 2 indivíduos, realizar crosover e mutação,
        # e adicionar o novo indivíduo à próxima geração

    return proxima_ger
```



O que devemos fazer

```
# O CÓDIGO PRINCIPAL

# Dica: nessa parte do código, você precisará
#       de várias das funções que você já implementou

num_geracoes = 100
jogo = DinoGame(fps=0)

# Crie a população usando populacao_aleatoria(NUM_INDIVIDUOS)
populacao = ?
```



O QUE DEVEMOS FAZER

```
# O CÓDIGO PRINCIPAL

for ger in range(num_geracoes):
    # Crie uma lista `fitness` com o fitness de cada indivíduo da população
    # (usando a função calcular_fitness e um `for` loop).

    # Atualize a população usando a função próxima_geração.
    fitness = ?
    populacao = ?

    # Opcional: parar se o fitness estiver acima de algum valor (p.ex. 300)

mostrar_melhor_individuo(jogo, populacao, fitness)

# Referência: for loop (for x in lista)
#             list.append()
```

OBRIGADO(A)!

Nos acompanhe:

fb.com/**grupoturing.poliusp**

medium.com/**turing-talks**