Universidade do Sul de Santa Catarina

Emilly Ruff de Camargo
Rafaela Martins coelho
Vitor Luiz Dalmuth
Giorgio Buka
Gabriel Jabour
Guilherme Izaias Fernandes

A3 ESTRUTURAs DE DADOS E ANÁLISE DE ALGORITMOS

DOCUMENTAÇÃO

Palhoça

2025

Introdução

Nome do Jogo: Jogo da Memória (Matching Game) - Implementado como "Jogo da Memória Visual Top"

O Jogo da Memória é uma atividade lúdica consagrada e de apelo universal, apreciada por diversas faixas etárias devido à sua mecânica simples, porém envolvente, e aos significativos benefícios cognitivos que proporciona. Fundamentalmente, o jogo atua como um estimulante para a concentração, exigindo que os jogadores mantenham o foco nas cartas reveladas e em suas posições; aprimora a percepção visual, à medida que os jogadores distinguem e comparam imagens ou símbolos; e, de forma proeminente, exercita a memória de curto prazo (ou memória de trabalho), que é a capacidade de reter e manipular informações por um breve período.

A premissa básica do Jogo da Memória envolve a identificação de pares de cartas idênticas que são inicialmente dispostas com as faces voltadas para baixo sobre uma superfície, configurando o que chamamos de tabuleiro. Os jogadores, em turnos alternados, revelam duas cartas por vez, com o objetivo de encontrar um par correspondente. No projeto "Jogo da Memória Visual Top", as cartas contêm perguntas e respostas sobre estruturas de dados, tornando-o uma ferramenta educativa interativa.

A popularidade do Jogo da Memória transcende o mero entretenimento. Ele é frequentemente e eficazmente utilizado como ferramenta pedagógica, auxiliando no desenvolvimento infantil em áreas como reconhecimento de formas, cores, números, letras e vocabulário. Educadores valorizam sua capacidade de ensinar conceitos de forma lúdica e interativa. Além disso, o jogo encontra aplicação em contextos terapêuticos, como na reabilitação cognitiva de adultos após lesões cerebrais ou no suporte a idosos para manutenção da acuidade mental e prevenção do declínio cognitivo associado à idade.

Uma das grandes vantagens do Jogo da Memória é sua versatilidade e adaptabilidade. A complexidade pode ser facilmente ajustada para diferentes públicos e objetivos:

- **Número de Cartas:** Um tabuleiro com poucas cartas (ex: 6 ou 8) é ideal para crianças pequenas, enquanto um número maior (ex: 30, 50 ou mais) oferece um desafio considerável para adultos. A implementação descrita neste documento utiliza um tabuleiro fixo de 3x4, totalizando 12 cartas (6 pares).
- **Natureza dos Pares:** As cartas podem apresentar desde simples figuras

geométricas ou cores para o público infantil, até conceitos mais abstratos, palavras em um idioma estrangeiro, datas históricas, fórmulas químicas, ou qualquer outro conteúdo que se deseje memorizar ou aprender. No jogo desenvolvido, os pares são formados por uma pergunta sobre estruturas de dados e sua respectiva resposta, cada par associado a uma cor de fundo distinta para facilitar a diferenciação visual.

- **Similaridade entre Imagens:** O grau de semelhança entre as imagens que *não* formam pares também pode aumentar a dificuldade, exigindo maior atenção aos detalhes.

Historicamente, jogos de pareamento existem há séculos em diversas culturas, mas a versão que conhecemos hoje, com cartas ilustradas e a dinâmica de virá-las, ganhou popularidade no século XX. A transição para o meio digital, impulsionada pelo avanço da tecnologia, expandiu exponencialmente suas possibilidades. Versões computadorizadas, como a desenvolvida em Python com a biblioteca Tkinter, permitem a incorporação de interfaces gráficas, interações de clique, e crucialmente para este projeto, facilitam a gestão complexa do estado do jogo através de estruturas de dados eficientes, como listas, filas e pilhas, que serão o foco da exploração neste documento. O jogo implementado é para dois jogadores e registra a pontuação, exibindo o vencedor ao final.

Dinâmica do Jogo da Memória

A dinâmica do Jogo da Memória é baseada na alternância de turnos entre os jogadores. No formato mais comum, e o adotado neste projeto, dois jogadores se revezam na tarefa de encontrar pares de cartas idênticas em um tabuleiro.

A Rodada Típica:
Em cada rodada, o jogador da vez possui o direito de revelar exatamente duas cartas do tabuleiro. A escolha dessas cartas é um momento de decisão estratégica, onde o jogador pode usar informações de jogadas anteriores ou, no início do jogo, contar com a sorte.

1. O jogador seleciona a primeira carta, clicando sobre ela na interface gráfica. A carta é virada para cima, expondo seu conteúdo (pergunta ou resposta sobre estrutura de dados) a todos os participantes.
2. Em seguida, o jogador seleciona a segunda carta da mesma forma.

Após a revelação da segunda carta, ocorre a verificação automática pelo sistema:

- **Par Encontrado (Sucesso):** Se as duas cartas reveladas formarem um par pergunta-resposta válido (conforme definido na estrutura pares do código), o jogador é recompensado. Ele marca um ponto (a variável pontuação do jogador é incrementada). As cartas que formam o par permanecem viradas para cima e são consideradas "resolvidas", não podendo ser selecionadas novamente. No jogo desenvolvido, o jogador que encontra um par joga novamente, mantendo seu turno.
- **Nenhum Par Encontrado (Insucesso):** Caso as cartas reveladas não formem um par válido, elas são viradas novamente com a face para baixo após um breve intervalo de tempo (800ms, controlado por root.after()), permitindo que os jogadores memorizem suas posições e conteúdos. Nenhum ponto é marcado. Após as cartas serem desviradas, a vez passa para o próximo jogador da sequência (a fila turnos é rotacionada).

À medida que pares são encontrados e efetivamente "saem do jogo" (não podem mais ser selecionados e seu estado muda em estado_tabuleiro), o número de opções virgens disponíveis diminui progressivamente. Isso tem um duplo efeito:

- Por um lado, simplifica a tarefa de memorização das cartas restantes e suas possíveis localizações.
- Por outro lado, aumenta a probabilidade de acerto nas jogadas subsequentes, pois a chance de virar um par por sorte ou por dedução lógica aumenta.

Isso geralmente acelera o ritmo da partida em suas fases finais. A tensão também pode aumentar, especialmente se a pontuação (pontuacao) estiver acirrada e restarem poucos pares. Cada jogada se torna mais crítica.

O jogo prossegue através desses ciclos de revelação, verificação e memorização até que todas as cartas tenham sido pareadas (condição verificada pela função todos_revelados()). O vencedor é aquele que acumulou o maior número de pares ao final da partida, conforme registrado na estrutura de pontuação. Em caso de igualdade, a partida é declarada um empate. Uma mensagem (messagebox.showinfo) exibe o resultado final.

Funcionamento

O Jogo da Memória, conhecido internacionalmente por nomes como "Concentration", "Memory Game", ou "Pairs", e no Brasil popularmente como "Jogo de Pares", é uma atividade lúdica de ampla difusão. Sua dinâmica, embora pareça simples à primeira vista, envolve uma interação fascinante entre a sorte e diversas habilidades cognitivas, baseando-se fundamentalmente na capacidade de memorização e na associação rápida de elementos visuais – no caso da implementação descrita, textos de perguntas e respostas.

Componentes e Configuração Inicial (Implementação Python/Tkinter):
A versão desenvolvida, "Jogo da Memória Visual Top", é composta por:

- **Um conjunto de cartas:** Definido pela estrutura pares_com_cores no código. Cada elemento principal é uma tupla contendo um par (pergunta, resposta) e uma string hexadecimal para a cor de fundo da carta. São 6 pares únicos, resultando em 12 cartas no total.
- **Um tabuleiro:** Representado graficamente pela biblioteca Tkinter, organizado em uma grade 3x4. As cartas são inicialmente embaralhadas (usando random.shuffle(cartas)) e distribuídas logicamente na estrutura tabuleiro (uma lista de listas). Visualmente, são botões que, ao serem clicados, revelam o texto da carta em um Label com a cor de fundo especificada.

No contexto deste trabalho, o jogo foi desenvolvido para dois jogadores ("Jogador1" e "Jogador2") que competem, alternando turnos, com o objetivo primordial de encontrar e coletar o maior número possível de pares pergunta-resposta corretos antes que todas as cartas sejam reveladas.

Mecânica de Jogo Aprofundada:
A mecânica é estruturada em turnos, gerenciados pela fila turnos. O jogador da vez realiza as seguintes ações através da interface gráfica:

1. **Seleção da Primeira Carta:** O jogador clica em um botão (carta virada para baixo) no tabuleiro. A função virar_carta(linha, coluna) é chamada. O botão é substituído por um Label exibindo o texto da carta e sua cor de fundo. As informações da carta (linha, coluna) são adicionadas à pilha.
2. **Seleção da Segunda Carta:** O jogador clica em outro botão. O mesmo processo ocorre. A pilha agora contém as coordenadas de duas cartas.
3. **Comparação Automática:** Se a pilha contém duas cartas, a função verificar_par() é chamada após um breve delay (800ms via root.after()).
    - **Se forem um par válido** (conforme definido em pares): A pontuação do jogador atual (obtido de turnos[0]) é incrementada. A pilha é limpa. O jogador mantém o turno para jogar novamente.

- **Se forem diferentes:** A vez passa para o próximo jogador (turnos.rotate(-1)). Após outro delay de 800ms, a função esconder_cartas() é chamada, revertendo os Labels para botões (cartas viradas para baixo) e limpando a pilha.

Este ciclo de tentativas, revelações e memorização prossegue até que todas as cartas do tabuleiro tenham sido pareadas (quando todos_revelados() retorna True).

Ao final da partida, o jogador com mais pares (pontuacao) vence. Uma messagebox exibe o resultado.

Fundamentos das Estruturas de Dados Essenciais para o Jogo

Antes de entrarmos na aplicação específica das estruturas de dados no Jogo da Memória desenvolvido, é crucial revisitar e solidificar os conceitos fundamentais de **listas, filas e pilhas**. Estas três estruturas são os pilares que sustentam a lógica e a funcionalidade implementadas neste projeto em Python. A escolha criteriosa e a aplicação correta de estruturas de dados são aspectos primordiais na ciência da computação e na engenharia de software, pois impactam diretamente a eficiência do algoritmo, a clareza do código, a facilidade de manutenção e a escalabilidade de qualquer sistema.

1. Listas (Python list):
Uma lista em Python é uma coleção ordenada e mutável de itens, onde cada item (ou elemento) possui uma posição específica, referenciada por um índice numérico (começando em 0). As listas Python são extremamente versáteis, permitindo o armazenamento de múltiplos valores de tipos de dados heterogêneos em uma única variável.

- **Operações Comuns Utilizadas no Projeto:**
  - Criação de listas (ex: cartas = [...], tabuleiro = [[]]).
  - Adicionar um item ao final (append(), usado para construir a pilha).
  - Acessar um item por seu índice (leitura, ex: tabuleiro[linha][coluna]).
  - Modificar um item em um determinado índice (ex: estado_tabuleiro[linha][coluna] = 'X').
  - Iterar (percorrer) sobre todos os itens da lista (usado na inicialização, verificação de fim de jogo, etc.).
  - Verificar o tamanho (len()).
  - List comprehensions para criação e transformação concisa de listas (usadas extensivamente na inicialização de pares, cartas, tabuleiro, estado_tabuleiro).
  - Embaralhamento (random.shuffle(), aplicado à lista cartas).
- **Implementação em Python:** As listas Python são implementadas como arrays dinâmicos, o que geralmente significa que o acesso a um elemento por índice é uma operação O(1). Inserções ou remoções no meio podem ser O(N), mas append() ao final é O(1) amortizado.

Para o Jogo da Memória, a capacidade de acessar diretamente uma carta pela sua "coordenadas" no tabuleiro (usando listas de listas para representar a grade 2D) e a facilidade de embaralhar seus elementos são propriedades extremamente valiosas que as listas Python oferecem.

2. Filas (Python collections.deque):

Uma fila é uma estrutura de dados linear que segue estritamente o princípio FIFO (First-In, First-Out). No projeto, a estrutura deque (double-ended queue) da biblioteca collections é usada para implementar a fila de turnos dos jogadores.

- **Operações Primárias Utilizadas no Projeto (com deque):**
  - append(item): Adiciona item ao final (direita) da fila (usado para recolocar o jogador no fim da fila de turnos).
  - popleft(): Remove e retorna o item do início (esquerda) da fila (poderia ser usado para pegar o jogador da vez, embora o código use turnos[0] para ver e turnos.rotate() para passar a vez).
  - rotate(n): Rotaciona os elementos da deque. turnos.rotate(-1) move o elemento da frente para o final, efetivamente passando o turno.
- **Operações Secundárias (Comuns):**
  - Acesso ao elemento da frente: turnos[0] é usado para identificar o jogador atual.
  - IsEmpty: Verificação implícita pelo fluxo do jogo (sempre há jogadores na fila).
  - Size: len(turnos).
- **Implementação:** collections.deque é implementada como uma lista duplamente encadeada, oferecendo desempenho O(1) para adições e remoções em ambas as extremidades.

No caso da implementação descrita, a deque é ideal para o controle de turnos em um jogo multiplayer (dois jogadores), onde cada jogador aguarda sua vez de forma ordenada.

3. Pilhas (Python list usada como Pilha):
Uma pilha é outra estrutura de dados linear, mas que opera sob o princípio LIFO (Last-In, First-Out). No código Python fornecido, uma lista (pilha = []) é usada para simular o comportamento de uma pilha.

- **Operações Fundamentais Utilizadas (com lista como pilha):**
  - append(item): Adiciona um novo item ao topo da pilha (equivalente a push). Usado para adicionar as coordenadas (linha, coluna) das cartas viradas.
  - pop(): Removeria e retornaria o item do topo (não usado explicitamente para remoção individual, mas a pilha é inspecionada e depois limpa com pilha.clear()).
  - Acesso aos elementos: Os elementos são acessados por desempacotamento (l1, c1), (l2, c2) = pilha quando len(pilha) == 2.
- **Operações Secundárias (Comuns):**
  - clear(): Remove todos os itens da lista/pilha (usado para limpar a pilha após uma jogada).

    - IsEmpty: Verificado por len(pilha).
    - Size: len(pilha) (usado para verificar se duas cartas foram viradas).
- **Implementação:** Usar uma lista Python como pilha é comum e eficiente para operações no "topo" (final da lista). append() e pop() (do final) são O(1) amortizado.

No Jogo da Memória, a pilha é ideal para gerenciar as duas cartas temporariamente viradas em uma jogada, permitindo fácil acesso para comparação e para saber quais reverter se não formarem um par.

A compreensão aprofundada dessas três estruturas de dados – suas operações, princípios de funcionamento (FIFO/LIFO), vantagens, desvantagens e casos de uso típicos – é fundamental para apreciar as decisões de design e implementação tomadas no desenvolvimento do Jogo da Memória, conforme será detalhado extensivamente nas seções subsequentes deste documento.

Justificativa e Utilização Estratégica das Estruturas de Dados no Jogo

Durante as fases de concepção e implementação do Jogo da Memória, a escolha das estruturas de dados não foi um ato arbitrário, mas sim um processo deliberado e crucial para garantir a lógica correta, a eficiência operacional e a clareza do código do sistema. Três estruturas de dados fundamentais foram empregadas, cada uma desempenhando um papel específico e essencial no funcionamento coeso do jogo: a fila (implementada com collections.deque), a pilha (implementada com list) e a lista (list). A aplicação sinérgica dessas estruturas contribuiu decisivamente para a organização das jogadas, o controle preciso das cartas reveladas em cada turno e a representação eficiente e manipulável do tabuleiro.

A seleção dessas estruturas baseou-se nas características intrínsecas de cada uma e em como elas se alinhavam perfeitamente com os requisitos funcionais e operacionais do jogo implementado em Python. Considerou-se como cada estrutura poderia otimizar o gerenciamento de turnos dos jogadores, o manuseio das cartas selecionadas em uma jogada (incluindo a possibilidade de reverter a ação) e a organização geral do tabuleiro de cartas.

Justificativa Detalhada para as Escolhas:

- **Fila (collections.deque) para Controle de Turnos:**
  - **Por que uma deque como Fila?** A natureza sequencial e inerentemente justa da alternância de turnos em um jogo para múltiplos jogadores encontra uma representação natural e elegante na estrutura de uma fila (FIFO). O collections.deque em Python é otimizado para adições e remoções em ambas as extremidades (O(1)), tornando-o ideal. O jogador atual é turnos[0], e turnos.rotate(-1) move eficientemente o jogador da frente para o final, passando o turno.
  - **Alternativas Consideradas e Descartadas:**
    - *Lista Python como Fila:* Usar list.pop(0) para desenfileirar é O(N), o que é menos eficiente que deque.popleft() ou deque.rotate(). Para apenas dois jogadores, a diferença de desempenho é negligível, mas deque é semanticamente mais correto e escalável.
  - **Conclusão da Escolha:** A collections.deque simplifica a lógica de alternância com rotate(), promove a justiça e oferece excelente desempenho.
- **Pilha (list) para Cartas da Rodada Atual:**

    - **Por que uma list como Pilha?** A necessidade de armazenar temporariamente as duas cartas que um jogador vira em uma rodada (no caso, suas coordenadas (linha, coluna)) e, crucialmente, a capacidade de processá-las e depois limpar o armazenamento para a próxima dupla é bem atendida por uma lista Python usada como pilha. As operações append() (para push) e clear() (ou acessar e depois limpar) são eficientes. O acesso às duas cartas é direto, pois a pilha só conterá no máximo dois elementos: (l1, c1), (l2, c2) = pilha.
    - **Conclusão da Escolha:** A lista Python oferece uma maneira simples e eficaz de gerenciar o estado transitório das duas cartas selecionadas, com append para adicionar e acesso direto aos dois únicos elementos esperados, seguido de clear.
- **Lista (list e list de list) para o Tabuleiro de Cartas e Estados:**
    - **Por que Listas?** A representação do tabuleiro de jogo, que contém os valores das cartas (tabuleiro), o estado de visibilidade (estado_tabuleiro), e as referências aos widgets da GUI (widgets), necessita de uma estrutura que permita acesso eficiente por índice e fácil manipulação. Listas Python, especialmente listas de listas para representar grades 2D, são ideais.
        - cartas: Lista 1D para embaralhamento inicial.
        - tabuleiro: Lista 2D (3x4) com os valores/textos das cartas.
        - estado_tabuleiro: Lista 2D (3x4) marcando cartas ocultas ('X') ou reveladas.
        - widgets: Lista 2D (3x4) armazenando os objetos Button/Label do Tkinter.
    - **Conclusão da Escolha:** Listas Python oferecem um equilíbrio ideal entre simplicidade de implementação, acesso indexado O(1) (essencial para uma grade), e uma rica gama de operações (slicing, list comprehensions, random.shuffle) usadas extensivamente na inicialização e manipulação do tabuleiro.

Nas seções subsequentes, explorar-se-á em profundidade como cada uma dessas estruturas – deque para filas, list para pilhas, e list para o tabuleiro – foi concretamente implementada e utilizada no código Python para construir a funcionalidade completa e robusta do "Jogo da Memória Visual Top".

A Fila (collections.deque) e Sua Aplicação Primordial no Controle de Turnos dos Jogadores

A **fila**, implementada no código Python através da estrutura collections.deque, foi estrategicamente empregada como o mecanismo central e mais intuitivo para o controle de turnos entre os dois jogadores no Jogo da Memória. Como uma estrutura de dados otimizada para adições e remoções em ambas as extremidades, a deque permite simular eficientemente o revezamento ordenado, justo e cíclico dos jogadores, seguindo a lógica FIFO (First-In, First-Out) de forma conceitual.

Implementação Detalhada e Funcionamento do Controle de Turnos com deque:

1. Inicialização da Fila de Turnos:
   Ao início de cada partida, a fila turnos é criada e populada com os identificadores dos dois jogadores:
   from collections import deque

   turnos = deque(["Jogador1", "Jogador2"])
   # 'Jogador1' está na frente (índice 0), 'Jogador2' está no final (índice 1).

   A estrutura pontuacao também é inicializada para registrar os pontos de cada jogador:
   pontuacao = {"Jogador1": 0, "Jogador2": 0}

2. Identificação do Jogador da Vez:
   A cada momento, o jogador ativo é aquele que está na frente da fila turnos. No código, isso é consistentemente identificado acessando o primeiro elemento:
   jogador_atual_id = turnos[0]
   # Ex: info_label.config(text=f"Turno de {turnos[0]} | Placar: {placar}")
   # Ex: pontuacao[turnos[0]] += 1

   Este acesso por índice [0] em uma deque é uma operação O(1).
3. Alternância de Turno – Rotação da Fila:
   A passagem do turno ocorre quando um jogador vira duas cartas que não formam um par. Neste caso, a fila turnos é rotacionada para a esquerda por uma posição usando turnos.rotate(-1). Isso move o jogador que estava na frente (turnos[0]) para o final da fila, e o próximo jogador automaticamente assume a posição da frente.

```
# Dentro da função verificar_par(), no bloco 'else' (par incorreto):
# turnos.rotate(-1)
```

A operação rotate(-1) em uma deque é eficiente (geralmente O(k) onde k é o número de rotações, aqui k=1, efetivamente O(Njogadores) que para 2 jogadores é O(1) na prática).

4. Manutenção do Turno em Caso de Acerto:
   Se o jogador atual encontra um par correto, a pontuação é atualizada, mas a fila turnos não é rotacionada. Isso significa que turnos[0] continua referenciando o mesmo jogador, permitindo que ele jogue novamente (turno bônus). Essa lógica está implementada na função verificar_par(): a rotação só ocorre no else.
5. Atualização da Informação Visual:
   A função atualizar_info() é chamada para refletir na interface gráfica (via info_label) qual jogador está ativo e o placar atual.
   ```
   def atualizar_info():
       placar = f"{pontuacao['Jogador1']} x {pontuacao['Jogador2']}"
       info_label.config(text=f"Turno de {turnos[0]}  |  Placar: {placar}")
   ```

Vantagens Detalhadas da Utilização da deque para Turnos no Código:

- **Eficiência:** Operações como append(), popleft(), acesso ao primeiro elemento [0], e rotate() são eficientes em deques. Para um número pequeno e fixo de jogadores (2), rotate(-1) é muito rápido.
- **Simplicidade e Clareza:** A lógica de verificar turnos[0] para o jogador atual e usar turnos.rotate(-1) para passar a vez é concisa e fácil de entender.
- **Justiça Inerente:** Garante que os jogadores se alternem corretamente.
- **Escalabilidade (Conceitual):** Embora o jogo seja para 2 jogadores, a deque lidaria bem com mais jogadores, mantendo a mesma lógica de rotação.

No contexto deste projeto específico, a collections.deque se mostrou uma escolha excelente e idiomática em Python para gerenciar o estado dos turnos de forma robusta e eficiente.

Pilha (list) e Seu Papel no Gerenciamento das Cartas Viradas na Rodada

A **pilha** (stack), no código Python fornecido, é implementada utilizando uma lista nativa do Python (pilha = []). Ela desempenha um papel fundamental e altamente eficiente ao armazenar temporariamente as informações das cartas que são viradas pelo jogador durante a sua rodada atual. Embora uma lista Python possa fazer muito mais, quando usada com operações como append() para adicionar ao final (topo) e pop() para remover do final (topo), ou simplesmente clear() e acesso por índice como no código, ela se comporta eficazmente como uma pilha LIFO (Last-In, First-Out) para este caso de uso específico onde no máximo duas cartas são mantidas.

Implementação Detalhada e Funcionamento do Gerenciamento de Cartas com a Lista como Pilha:

1. Estado Inicial da Pilha da Jogada:
   A variável global pilha é inicializada como uma lista vazia:
   pilha = []

   No início de cada nova seleção de um par de cartas por um jogador (ou seja, quando len(pilha) é 0 ou 1), esta pilha armazena as seleções.
2. Seleção e "Empilhamento" da Carta Virada:
   Quando um jogador clica em uma carta no tabuleiro e a função virar_carta(linha, coluna) é executada, as coordenadas da carta virada são adicionadas ao final da lista pilha (agindo como um push):
   # Dentro da função virar_carta(linha, coluna):
   # pilha.append((linha, coluna))

   A pilha agora contém uma tupla (linha, coluna). Se outra carta for virada, outra tupla será adicionada. A pilha nunca conterá mais de duas tuplas, pois a lógica em virar_carta impede a seleção de uma terceira carta se len(pilha) == 2.
3. Acesso e Comparação das Cartas:
   Quando duas cartas foram viradas (len(pilha) == 2), a função verificar_par() é chamada. Dentro dela, as informações das duas cartas são acessadas diretamente da pilha por desempacotamento, aproveitando que se sabe que há exatamente dois elementos:
   # Dentro da função verificar_par():
   # (l1, c1), (l2, c2) = pilha
   # # l1, c1 são as coordenadas da primeira carta virada
   # # l2, c2 são as coordenadas da segunda carta virada

t1 = tabuleiro[l1][c1] # Valor da primeira carta
t2 = tabuleiro[l2][c2] # Valor da segunda carta

4. **Resultado da Comparação e Ações Subsequentes:**
    - **Se as cartas forem iguais (par encontrado):**
        - A pontuação do jogador (pontuacao[turnos[0]]) é incrementada.
        - A pilha é limpa para a próxima jogada do mesmo jogador (já que ele joga novamente):
          # Dentro de verificar_par(), se par_correto:
          # pilha.clear()

    - **Se as cartas forem diferentes (não formam par):**
        - O turno é passado para o próximo jogador (turnos.rotate(-1)).
        - A função esconder_cartas() é agendada (root.after(800, esconder_cartas)).
        - Dentro de esconder_cartas(), os elementos da pilha (as coordenadas das duas cartas erradas) são usados para reverter seus visuais para botões ocultos. Após isso, a pilha é limpa:
          # Dentro da função esconder_cartas():
          # for linha, coluna in pilha:
          #   widgets[linha][coluna].destroy()
          #   btn = criar_botao(linha, coluna)
          #   widgets[linha][coluna] = btn
          #   estado_tabuleiro[linha][coluna] = 'X'
          # pilha.clear()

Vantagens Detalhadas da Utilização da Lista Python como Pilha Neste Contexto:

- **Simplicidade:** Usar uma lista Python para uma pilha de tamanho fixo e pequeno (máximo 2 elementos) é muito simples e direto.
- **Eficiência para Operações Utilizadas:** append() é O(1) amortizado. clear() é eficiente. Acessar os dois elementos por desempacotamento é O(1) porque a lista é pequena.
- **Controle Preciso da Rodada:** Isola as coordenadas das duas cartas da jogada atual.
- **Reversibilidade Facilitada:** As coordenadas na pilha são usadas diretamente por esconder_cartas para saber quais widgets reverter.

A implementação da pilha como uma lista no código é uma solução pragmática e

eficiente para as necessidades específicas do Jogo da Memória, onde o "histórico" da pilha é sempre de no máximo duas ações (virar duas cartas).

Lista (list) e Versatilidade na Representação e Manipulação do Tabuleiro

A estrutura de dados **lista** do Python é utilizada de maneira extensiva e central no "Jogo da Memória Visual Top" para representar e gerenciar os diversos aspectos do tabuleiro e do estado do jogo. Sua flexibilidade, capacidade de armazenar coleções ordenadas e, crucialmente, de suportar aninhamento (listas de listas para grades 2D), a tornam ideal para esta aplicação.

Implementação e Representação Detalhada do Tabuleiro com Listas Python:

1. Definição Inicial dos Pares e Cores (pares_com_cores):
   Uma lista de tuplas é usada para definir os pares de pergunta-resposta e suas cores associadas:
   ```
   pares_com_cores = [
      (("Qual estrutura adiciona no final e remove do início?", "Fila"), "#add8e6"),
      # ... outros 5 pares ...
   ]
   ```

   Esta lista (list) é a fonte primária para o conteúdo das cartas.
2. Extração dos Pares Lógicos (pares):
   Uma list comprehension é usada para criar uma lista contendo apenas os pares (pergunta, resposta):
   ```
   pares = [p for (p, _) in pares_com_cores]
   # Resultado: [("Pergunta1", "Resposta1"), ("Pergunta2", "Resposta2"), ...]
   ```

   Esta lista pares é usada posteriormente na função verificar_par para checar se as duas cartas viradas constituem um par válido.
3. Mapeamento de Texto para Cor (texto_para_cor):
   Um dicionário (que não é uma lista, mas é derivado dela) é criado para mapear cada string de pergunta ou resposta à sua cor de fundo, facilitando a estilização das cartas reveladas:
   ```
   texto_para_cor = {texto: cor for (par_tupla, cor) in pares_com_cores for texto in par_tupla}
   # Ex: {"Pergunta1": "#cor1", "Resposta1": "#cor1", ...}
   ```

4. Criação da Lista Unidimensional de Todas as Cartas (cartas):
   Uma list comprehension "achata" os pares em uma única lista contendo todas as strings (perguntas e respostas) que irão para o tabuleiro. Cada pergunta e cada resposta aparecem individualmente.

cartas = [item for par_tupla in pares for item in par_tupla]
# Resultado: ["Pergunta1", "Resposta1", "Pergunta2", "Resposta2", ..., "Pergunta6", "Resposta6"]
# Esta lista terá 12 elementos.

5. Embaralhamento da Lista de Cartas (random.shuffle):
   A lista cartas é embaralhada para garantir a aleatoriedade da disposição no tabuleiro:
   random.shuffle(cartas)

6. Criação do Tabuleiro Lógico Bidimensional (tabuleiro):
   A lista 1D cartas embaralhada é transformada em uma lista de listas (uma matriz 3x4) que representa a disposição lógica das cartas no tabuleiro:
   tabuleiro = [cartas[i:i+4] for i in range(0, 12, 4)]
   # Ex: tabuleiro[0][0] contém o texto da carta na primeira linha, primeira coluna.

   Esta estrutura tabuleiro armazena os *valores* (textos) reais das cartas em suas posições.
7. Criação do Estado de Visibilidade do Tabuleiro (estado_tabuleiro):
   Uma lista de listas paralela, de mesmas dimensões (3x4), é criada para rastrear o estado de cada carta (oculta ou revelada). Inicialmente, todas estão ocultas, marcadas com 'X':
   estado_tabuleiro = [['X'] * 4 for _ in range(3)]
   # Ex: estado_tabuleiro[linha][coluna] é 'X' se oculta, ou o texto da carta se revelada.

   Esta estrutura é modificada quando uma carta é virada (em virar_carta) e quando é escondida novamente (em esconder_cartas). A função todos_revelados() verifica esta lista para determinar o fim do jogo.
8. Armazenamento das Referências aos Widgets da GUI (widgets):
   Outra lista de listas (3x4) é usada para manter referências aos objetos da interface gráfica (Tkinter Button ou Label) que representam cada carta na tela:
   widgets = [[None for _ in range(4)] for _ in range(3)]

   Quando um botão Button é criado por criar_botao(linha, coluna), a referência é armazenada em widgets[linha][coluna]. Quando uma carta é virada, o Button é destruído e substituído por um Label, e a referência em widgets é atualizada para o novo Label. Isso permite que as funções esconder_cartas e virar_carta modifiquem diretamente os elementos visuais corretos.

Vantagens da Utilização de Listas Python para o Tabuleiro:

- **Representação Natural de Grades:** Listas de listas são uma forma intuitiva e direta de representar grades bidimensionais como um tabuleiro de jogo.
- **Acesso Indexado Eficiente:** O acesso a tabuleiro[l][c], estado_tabuleiro[l][c], e widgets[l][c] é O(1).
- **Flexibilidade e Poder das List Comprehensions:** A criação e transformação inicial das listas de cartas e do tabuleiro são feitas de forma concisa e eficiente usando list comprehensions.
- **Operações Nativas:** Funções como random.shuffle() operam diretamente sobre listas.

As múltiplas listas (1D e 2D) são usadas de forma coordenada para gerenciar de forma robusta os dados, o estado e a representação visual do tabuleiro do Jogo da Memória.

Aprofundamento no Código de Manipulação do Tabuleiro (Estrutura de Lista em Python)

Para solidificar a compreensão de como a estrutura de **lista** do Python é central na manipulação do tabuleiro no "Jogo da Memória Visual Top", vamos detalhar os principais trechos de código da inicialização e as funções que interagem com essas listas.

1. Inicialização das Estruturas de Dados do Tabuleiro:
O código começa definindo os pares de pergunta-resposta e suas cores:

```
pares_com_cores = [
    (("Qual estrutura adiciona no final e remove do início?", "Fila"), "#add8e6"),
    (("Qual estrutura só usa o topo para adicionar e remover?", "Pilha"), "#dda0dd"),
    (("Qual estrutura pode guardar elementos em ordem?", "Lista"), "#90ee90"),
    (("Qual estrutura é usada para criar grades como o tabuleiro?", "Lista 2D"),
"#ffcccb"),
    (("Como é o comportamento da pilha?", "LIFO"), "#ffffcc"),
    (("Como é o comportamento da fila?", "FIFO"), "#ffb347")
]

# Lista de tuplas (pergunta, resposta)
pares = [p for (p, _) in pares_com_cores]

# Dicionário para mapear texto da carta para sua cor de fundo
texto_para_cor = {texto: cor for (par_tupla, cor) in pares_com_cores for texto in
par_tupla}

# Lista 1D com todos os textos das cartas (12 textos)
cartas = [item for par_tupla in pares for item in par_tupla]
random.shuffle(cartas) # Embaralha a lista 1D

# Lista 2D (3x4) representando o tabuleiro lógico com os textos das cartas
tabuleiro = [cartas[i:i+4] for i in range(0, 12, 4)]

# Lista 2D (3x4) representando o estado de visibilidade (inicialmente todas 'X' -
ocultas)
estado_tabuleiro = [['X'] * 4 for _ in range(3)]

# Lista 2D (3x4) para armazenar referências aos widgets Tkinter (botões/labels)
```

```
widgets = [[None for _ in range(4)] for _ in range(3)]
```

Esta sequência demonstra o uso de listas e list comprehensions para configurar de forma eficiente todas as estruturas de dados necessárias para o tabuleiro a partir de uma definição inicial (pares_com_cores).

2. Função criar_botao(linha, coluna):
Esta função é responsável por criar o widget Tkinter Button inicial para cada carta (quando está virada para baixo).

```
def criar_botao(linha, coluna):
    b = tk.Button(frame_tabuleiro, text="", bg="gray", fg="black", relief=tk.RAISED,
                  font=("Arial", 10, "bold"), wraplength=100,
                  command=lambda l=linha, c=coluna: virar_carta(l, c))
    b.grid(row=linha, column=coluna, sticky="nsew", padx=4, pady=4)
    return b # Retorna a referência ao botão criado
```

Na inicialização da interface, um loop preenche a lista widgets com os botões criados:

```
for i in range(3): # 3 linhas
    frame_tabuleiro.grid_rowconfigure(i, weight=1)
    for j in range(4): # 4 colunas
        frame_tabuleiro.grid_columnconfigure(j, weight=1)
        widgets[i][j] = criar_botao(i, j)
```

3. Função virar_carta(linha, coluna):
Esta função é chamada quando um botão (carta oculta) é clicado.

```
def virar_carta(linha, coluna):
    # Não permite virar se a carta já está revelada ou se já há 2 cartas na pilha
    if estado_tabuleiro[linha][coluna] != 'X' or len(pilha) == 2:
        return

    texto = tabuleiro[linha][coluna] # Pega o texto da carta do tabuleiro lógico
    estado_tabuleiro[linha][coluna] = texto # Atualiza o estado para revelado (com seu texto)

    widgets[linha][coluna].destroy() # Remove o widget Button antigo
```

```
    cor = texto_para_cor.get(texto, "#add8e6") # Pega a cor associada ao texto

    # Cria um Label para exibir o texto da carta revelada
    lbl = tk.Label(frame_tabuleiro, text=texto, bg=cor, fg="black",
                   font=("Arial", 10, "bold"), wraplength=100, justify="center",
                   relief=tk.RAISED, borderwidth=2)
    lbl.grid(row=linha, column=coluna, sticky="nsew", padx=4, pady=4)
    widgets[linha][coluna] = lbl # Atualiza a referência na lista widgets para o novo
Label

    pilha.append((linha, coluna)) # Adiciona as coordenadas da carta virada à pilha

    if len(pilha) == 2: # Se duas cartas foram viradas
        root.after(800, verificar_par) # Agenda a verificação do par após 800ms
```

Esta função demonstra a interação crucial entre as listas tabuleiro (para obter o valor), estado_tabuleiro (para verificar e atualizar o estado de visibilidade) e widgets (para manipular a GUI).

4. Função esconder_cartas():
Chamada quando um par incorreto é formado, para virar as cartas de volta.

```
def esconder_cartas():
    for linha, coluna in pilha: # Itera sobre as coordenadas das duas cartas na pilha
        widgets[linha][coluna].destroy() # Remove o Label que mostrava a carta
        btn = criar_botao(linha, coluna) # Cria um novo Button (carta oculta)
        widgets[linha][coluna] = btn     # Atualiza a referência em widgets
        estado_tabuleiro[linha][coluna] = 'X' # Marca o estado como oculta ('X')
    pilha.clear() # Limpa a pilha para a próxima jogada
```

Novamente, pilha, widgets e estado_tabuleiro trabalham em conjunto.

5. Função verificar_par():
Verifica se as duas cartas na pilha formam um par.

```
def verificar_par():
    (l1, c1), (l2, c2) = pilha # Desempacota as coordenadas das duas cartas
    t1 = tabuleiro[l1][c1]   # Texto da primeira carta
    t2 = tabuleiro[l2][c2]   # Texto da segunda carta
```

```
    # Verifica se (t1, t2) ou (t2, t1) existe na lista 'pares'
    par_correto = any((t1, t2) == p or (t2, t1) == p for p in pares)

    if par_correto:
        pontuacao[turnos[0]] += 1 # Incrementa pontuação
        pilha.clear() # Limpa a pilha, jogador joga de novo
    else:
        turnos.rotate(-1) # Passa o turno
        root.after(800, esconder_cartas) # Agenda para esconder as cartas

    atualizar_info()
    if todos_revelados(): # Verifica se todas as cartas em 'estado_tabuleiro' não são 'X'
        fim_de_jogo()
```

Esta função lê da lista tabuleiro e da lista pares para tomar decisões.

O uso coordenado dessas listas (tabuleiro, estado_tabuleiro, widgets, pares) e do dicionário (texto_para_cor) é o que dá vida à lógica e à apresentação do Jogo da Memória. As list comprehensions agilizam a inicialização, e o acesso indexado permite a manipulação eficiente da grade.

Análise Aprofundada de Complexidade de Algoritmos (Notação Big O) no Jogo da Memória

Um requisito fundamental e de grande importância acadêmica neste projeto é a **análise da complexidade de tempo (utilizando a notação Big O)** de, no mínimo, uma das operações críticas implementadas que utilizam as estruturas de dados (listas, filas ou pilhas). A notação Big O é uma linguagem matemática usada em ciência da computação para descrever o comportamento assintótico (limitante) de uma função quando seu argumento tende a um valor particular ou ao infinito. Em termos práticos, ela ajuda a classificar algoritmos de acordo com como seu tempo de execução (ou, alternativamente, requisitos de espaço de memória) cresce à medida que o tamanho da entrada do problema aumenta. Compreender a complexidade Big O é vital para escrever software eficiente e escalável.

Analisam-se algumas operações chave no contexto do "Jogo da Memória Visual Top", considerando N_pares como o número de pares únicos de cartas (6 no caso), o que implica um total de M = 2 * N_pares = 12 cartas no tabuleiro.

1. Inicialização Completa do Tabuleiro:
Esta é uma operação realizada uma vez no início de cada partida.

- **Criação de pares, texto_para_cor, cartas:** A iteração sobre pares_com_cores (de tamanho N_pares) para criar essas estruturas é proporcional a N_pares. Portanto, O(Npares).
- **Embaralhamento da lista cartas (random.shuffle(cartas)):** cartas tem M elementos. O algoritmo de Fisher-Yates é O(M), que é O(2Npares), simplificando para O(Npares).
- **Distribuição das M cartas na grade 2D tabuleiro:** Envolve iterar sobre a lista cartas uma vez. Complexidade O(M) ou O(Npares).
- **Criação de estado_tabuleiro e widgets:** Inicializar estas matrizes de M posições é O(M) ou O(Npares).
- **Conclusão para a Inicialização Completa:** A complexidade dominante para toda a fase de inicialização do tabuleiro é O(Npares). Dado que N_pares é fixo (6), todas essas operações são efetivamente O(1) em termos de crescimento com uma entrada variável, mas é útil analisar em termos de N_pares para generalização.

2. Operações da Função virar_carta(linha, coluna):

- **Acesso a estado_tabuleiro[linha][coluna] e tabuleiro[linha][coluna]:** O(1).

- **len(pilha):** O(1).
- **widgets[linha][coluna].destroy():** Depende da implementação do Tkinter, mas geralmente rápido para um widget.
- **texto_para_cor.get(texto, ...):** Acesso a dicionário é, em média, O(1).
- **Criação de tk.Label e lbl.grid():** Criação e posicionamento de widget.
- **pilha.append((linha, coluna)):** O(1) amortizado para lista Python.
- **root.after(...):** Agendamento de evento, O(1).
- **Conclusão para virar_carta:** Dominada por operações O(1), então O(1).

3. Operações da Função verificar_par():

- **Desempacotamento da pilha:** O(1) (pois pilha tem sempre 2 elementos).
- **Acesso a tabuleiro:** O(1).
- **Loop any(... for p in pares):** Itera sobre a lista pares (tamanho N_pares). No pior caso, verifica todos os N_pares. Portanto, O(Npares).
- **pontuacao[turnos[0]] += 1:** Acesso a dicionário e incremento, O(1).
- **pilha.clear():** O(k) onde k é o tamanho da pilha (2 aqui), então O(1).
- **turnos.rotate(-1):** Em uma deque de N_jogadores elementos, rotate(k) é O(k×Njogadores) se implementado naively, ou O(Njogadores) para k rotações simples. Para deque, é mais eficiente, geralmente O(k) onde k é o número de elementos rotacionados, que é no máximo o tamanho da deque. Para rotate(-1) em uma deque de 2 jogadores, é O(1) efetivamente.
- **root.after(...):** O(1).
- **atualizar_info():** Envolve acesso a turnos[0] (O(1)) e pontuacao (O(1)), e info_label.config(), que é O(1).
- **todos_revelados():** Itera sobre estado_tabuleiro (M elementos). Portanto, O(M) ou O(Npares).
- **fim_de_jogo():** Envolve acesso a pontuacao e messagebox.showinfo(), O(1).
- **Conclusão para verificar_par:** A operação dominante é o loop para checar o par (any(...)) que é O(Npares) e todos_revelados() que é O(Npares). Portanto, verificar_par é O(Npares).

4. Operações da Função esconder_cartas():

- **Loop for linha, coluna in pilha:** Itera 2 vezes (tamanho da pilha). O(1).
- Dentro do loop: destroy(), criar_botao(), grid(), acesso a widgets e estado_tabuleiro são todos O(1).
- **pilha.clear():** O(1).
- **Conclusão para esconder_cartas:** O(1).

Análise Detalhada de uma Operação Específica: Verificação de Par (verificar_par)
A operação crítica mais complexa (em termos de N_pares) dentro do ciclo de jogo é

verificar_par, especificamente a linha:
par_correto = any((t1, t2) == p or (t2, t1) == p for p in pares)
Aqui, pares é uma lista de tuplas (pergunta, resposta), com N_pares (6) elementos. No pior caso, a expressão (t1, t2) == p or (t2, t1) == p será avaliada para cada um dos N_pares elementos em pares até que uma correspondência seja encontrada ou a lista termine. Cada comparação de tuplas e strings internas leva um tempo proporcional ao comprimento das strings, mas pode-se considerar isso como uma constante para uma dada pergunta/resposta. Portanto, esta parte da verificação é O(Npares).
A chamada a todos_revelados() também é O(M) ou O(Npares) pois itera sobre todas as M (12) posições do estado_tabuleiro.
Assim, a complexidade geral de verificar_par() é O(Npares).
Importância da Análise de Big O no Contexto do Jogo:
Para o tamanho fixo do tabuleiro (6 pares), O(Npares) é efetivamente O(1) (constante). O jogo será rápido e responsivo. Se o jogo fosse projetado para tabuleiros muito grandes (ex: N_pares = 100), a verificação do par e do fim de jogo poderiam começar a ter um impacto perceptível, embora linear ainda seja geralmente aceitável para interações do usuário. A maioria das outras interações diretas do jogador (clicar, virar) são O(1), o que é ideal.

Instruções Detalhadas e Sequenciais para Executar o Jogo/Simulador ("Jogo da Memória Visual Top")

Para executar e jogar o "Jogo da Memória Visual Top" implementado em Python com Tkinter, siga os passos abaixo. Estas instruções assumem que Python está instalado no sistema com a biblioteca Tkinter (que geralmente vem inclusa na instalação padrão do Python).

Fase 1: Preparação e Execução do Código

1. **Obtenha o Código:**
   - Certifique-se de ter o arquivo Python do jogo (ex: jogo_da_memoria.py) salvo em seu computador.
2. **Execute o Script Python:**
   - Abra um terminal ou prompt de comando.
   - Navegue até o diretório onde o arquivo jogo_da_memoria.py foi salvo.
   - Execute o jogo digitando: python jogo_da_memoria.py (ou python3 jogo_da_memoria.py dependendo da configuração do sistema).
   - Uma janela gráfica intitulada "Jogo da Memória Visual Top" com dimensões aproximadas de 960x540 pixels deverá aparecer.

Fase 2: Interagindo com o Jogo

3. **Interface Inicial:**
   - Visualizar-se-á um tabuleiro de cartas 3x4 (12 cartas no total). Todas as cartas estarão inicialmente viradas para baixo, exibidas como botões cinzas.
   - No topo da janela, uma etiqueta (info_label) mostrará: "Turno de Jogador1 | Placar: 0 x 0".
4. **Jogando uma Rodada (Turno do Jogador1):**
   - **Jogador1 escolhe a primeira carta:** Clique em qualquer um dos botões cinzas no tabuleiro.
     - O botão clicado será substituído por um Label colorido, revelando o texto da carta (uma pergunta ou resposta sobre estruturas de dados).
     - As coordenadas desta carta são adicionadas à pilha interna do jogo.
   - **Jogador1 escolhe a segunda carta:** Clique em outro botão cinza.
     - Este botão também se transformará em um Label colorido revelando seu texto.
     - As coordenadas desta segunda carta são adicionadas à pilha.
   - **Verificação Automática do Par:** Após um breve delay de 800 milissegundos

(controlado por root.after(800, verificar_par)), o sistema verificará se as duas cartas formam um par pergunta-resposta correto:

- **Se for um par correto:**
  - A pontuação do Jogador1 (em pontuacao) será incrementada em 1.
  - A info_label será atualizada com o novo placar.
  - As duas cartas permanecerão viradas para cima.
  - A pilha interna é limpa.
  - **Jogador1 joga novamente** (o turno não passa para Jogador2). Retorne ao passo "Jogador1 escolhe a primeira carta".
- **Se NÃO for um par correto:**
  - Após outro delay de 800 milissegundos (controlado por root.after(800, esconder_cartas)), as duas cartas reveladas voltarão a ser botões cinzas (viradas para baixo).
  - A pilha interna é limpa.
  - O turno passará para Jogador2 (a fila_turnos é rotacionada com turnos.rotate(-1)).
  - A info_label será atualizada para "Turno de Jogador2 | Placar: [placar atual]".

5. **Jogando uma Rodada (Turno do Jogador2):**
   - O processo é idêntico ao do Jogador1. Jogador2 clica em duas cartas, e o sistema verifica o par, atualiza o placar e determina quem joga em seguida.
6. **Continuando o Jogo:**
   - Os jogadores se alternam conforme descrito acima, tentando encontrar todos os 6 pares de cartas.
   - Cartas que já fazem parte de um par encontrado permanecem reveladas e não podem ser selecionadas novamente (a função virar_carta impede a seleção de cartas que não estão no estado 'X').

Fase 3: Fim de Jogo

7. **Condição de Fim de Jogo:**
   - O jogo termina quando todas as 12 cartas tiverem sido reveladas e pareadas (a função todos_revelados() retorna True ao verificar que não há mais 'X' em estado_tabuleiro).
8. **Exibição do Resultado:**
   - Uma caixa de mensagem (messagebox.showinfo) aparecerá com o título "Fim do Jogo".
   - A mensagem exibirá o placar final (ex: "Placar final: 4 x 2") e o resultado:
     - "Jogador1 venceu!" se Jogador1 tiver mais pontos.
     - "Jogador2 venceu!" se Jogador2 tiver mais pontos.

        - "Empate!" se as pontuações forem iguais.
    - Após clicar em "OK" na caixa de mensagem, a janela principal do jogo será fechada (root.quit()).

Estas instruções devem permitir que qualquer pessoa com o código e Python execute e compreenda o fluxo do "Jogo da Memória Visual Top".

Detalhamento das Funcionalidades Implementadas e Requisitos Técnicos Atendidos

Este projeto do Jogo da Memória, "Jogo da Memória Visual Top", foi concebido e desenvolvido com o objetivo central de aplicar, de forma prática e significativa, os conceitos fundamentais de estruturas de dados (com foco em listas, filas e pilhas) e os princípios de análise de algoritmos, utilizando Python e a biblioteca Tkinter para a interface gráfica, conforme delineado nos objetivos da disciplina. A seguir, apresenta-se um detalhamento de como cada um dos requisitos técnicos obrigatórios foi especificamente atendido na implementação do jogo.

1. **Utilização de Listas para Organização Dinâmica de Elementos:**
    - **Requisito Especificado:** "Utilizar listas para organizar elementos dinâmicos do jogo (jogadores, recursos, áreas, pontuações etc.)."
    - **Aplicação Concreta no Projeto:**
        - pares_com_cores: Uma **lista** de tuplas define os dados brutos das cartas (par pergunta-resposta e cor).
        - pares: Uma **lista** de tuplas (pergunta, resposta) derivada da anterior, usada para verificar se um par foi formado.
        - cartas: Uma **lista** 1D contendo todas as strings de perguntas e respostas, usada para o embaralhamento inicial com random.shuffle().
        - tabuleiro: Uma **lista de listas** (grade 2D, 3x4) que armazena a disposição lógica dos textos das cartas após o embaralhamento. É a "verdade" sobre qual texto está em qual posição.
        - estado_tabuleiro: Uma **lista de listas** (grade 2D, 3x4) paralela ao tabuleiro, que armazena o estado de cada carta ('X' para oculta, ou o texto da carta se revelada e ainda não parte de um par totalmente processado).
        - widgets: Uma **lista de listas** (grade 2D, 3x4) que armazena as referências aos objetos Button ou Label do Tkinter que representam visualmente cada carta na interface.
        - pilha: Embora conceitualmente uma pilha, é implementada como uma **lista** Python (pilha = []) para armazenar as coordenadas (linha, coluna) das duas cartas viradas na jogada atual.
2. **Utilização de Filas para Controle de Sequência (Turnos):**
    - **Requisito Especificado:** "Utilizar filas para controle de turnos, ordens de chegada ou tarefas."
    - **Aplicação Concreta no Projeto:** Uma **fila** é implementada usando collections.deque para a variável turnos.
        - turnos = deque(["Jogador1", "Jogador2"]) inicializa a fila com os dois

jogadores.
    - O jogador da vez é identificado por turnos[0].
    - A passagem de turno (quando um par não é formado) é realizada com turnos.rotate(-1), que move o jogador da frente para o final da fila de forma eficiente.
3. **Utilização de Pilhas para Ações Reversíveis e Estado Temporário:**
    - **Requisito Especificado:** "Utilizar pilhas para representar ações reversíveis, histórico de decisões ou uso de cartas/eventos."
    - **Aplicação Concreta no Projeto:** Uma **pilha**, implementada como uma lista Python (pilha = []), é usada para armazenar temporariamente as coordenadas (linha, coluna) das duas cartas que o jogador virou na rodada atual.
        - pilha.append((linha, coluna)) adiciona uma carta virada.
        - Quando len(pilha) == 2, as coordenadas são usadas para verificar o par.
        - Se não for um par, a função esconder_cartas() utiliza as coordenadas na pilha para identificar quais widgets na interface devem ser revertidos (Labels para Buttons) e quais posições no estado_tabuleiro devem voltar para 'X'. A pilha é então limpa com pilha.clear().
4. **Realização de Análise de Complexidade de Tempo (Notação Big O):**
    - **Requisito Especificado:** "Realizar a análise de complexidade de tempo (notação Big O) de ao menos uma das operações cruciais implementadas utilizando listas, filas ou pilhas. Esta análise deve ser incluída na documentação e apresentada durante a demonstração."
    - **Aplicação Concreta no Projeto:** A página 25 desta documentação apresenta uma análise de complexidade. A operação de verificar_par() (que envolve iteração sobre a lista pares) foi analisada como O(Npares), e o embaralhamento (random.shuffle sobre a lista cartas) como O(M) ou O(Npares).
5. **Uso das Ferramentas de Gerenciamento de Projeto (Trello e Git/GitHub):**
    - **Requisitos Especificados:** "Utilizar a plataforma Trello para o gerenciamento das atividades do projeto." e "Utilizar o sistema de controle de versão Git para o desenvolvimento colaborativo do código."
    - **Aplicação Concreta no Projeto:** Conforme detalhado anteriormente, estas ferramentas foram utilizadas para organização e versionamento.
6. **Mínimo de Dois Jogadores Implementado:**
    - **Requisito Especificado:** "O jogo deve ter no mínimo dois jogadores..."
    - **Aplicação Concreta no Projeto:** O jogo é explicitamente para "Jogador1" e "Jogador2", gerenciados pela fila turnos e com pontuações individuais em pontuacao.
7. **Uso Simultâneo e Integrado das Três Estruturas de Dados:**

- **Requisitos Especificados:** "A implementação tem que ser feita utilizando estruturas de dados." e "O Jogo deve utilizar as três estruturas de dados ao mesmo tempo."
- **Aplicação Concreta no Projeto:** O jogo utiliza ativamente e de forma interdependente:
  - **Listas:** Para pares_com_cores, pares, cartas, tabuleiro, estado_tabuleiro, widgets.
  - **Filas (deque):** Para turnos.
  - Pilhas (lista como pilha): Para pilha (cartas viradas na jogada).
    A funcionalidade do jogo emerge da interação coordenada dessas estruturas.

Adicionalmente, o código (Jogo da Memoria.py) é funcional, implementa a lógica do Jogo da Memória com uma interface gráfica em Tkinter, e a temática educativa sobre estruturas de dados é clara no conteúdo das cartas.

Desafios Encontrados Durante o Desenvolvimento e Soluções

Adotadas

O processo de desenvolvimento do "Jogo da Memória Visual Top" em Python com Tkinter, embora resultando em uma aplicação funcional e educativa, apresentou desafios inerentes ao desenvolvimento de software, especialmente com interfaces gráficas e lógica de jogo interativa. A superação desses obstáculos foi uma parte crucial do aprendizado.

1. **Desafio: Sincronização entre Lógica de Jogo e Atualizações da GUI Tkinter.**
    - **Descrição do Problema:** Garantir que as mudanças no estado lógico do jogo (ex: estado_tabuleiro, pilha, turnos, pontuacao) fossem refletidas corretamente e em tempo hábil na interface gráfica Tkinter (atualização de info_label, destruição de Button e criação de Label em widgets, etc.) foi um ponto central. A natureza orientada a eventos do Tkinter requer um manejo cuidadoso das atualizações da UI.
    - **Solução Adotada:**
        - Funções específicas como atualizar_info() foram criadas para centralizar as atualizações de partes da UI (como o placar e o jogador da vez).
        - As modificações nos widgets do tabuleiro (de Button para Label em virar_carta, e de volta para Button em esconder_cartas) foram feitas programaticamente, atualizando a lista widgets com as novas referências.
        - O uso de root.after(delay_ms, callback_function) foi essencial para introduzir pequenos delays (800ms) antes de esconder cartas não pareadas ou antes de processar o par. Isso permite que o jogador veja a segunda carta antes da verificação ou antes que ela seja escondida, melhorando a jogabilidade. Sem isso, as ações poderiam parecer instantâneas e confusas.
2. **Desafio: Gerenciamento do Estado das Cartas Viradas e Lógica de Pares.**
    - **Descrição do Problema:** Controlar corretamente quais cartas estavam viradas (usando a pilha para no máximo duas), como compará-las com a lista pares (que contém tuplas (pergunta, resposta)), e como lidar com a ordem (pergunta-resposta vs. resposta-pergunta) exigiu atenção.
    - **Solução Adotada:**
        - A pilha (lista Python) armazena tuplas (linha, coluna). A lógica em virar_carta impede que mais de duas cartas sejam adicionadas.
        - Em verificar_par, as coordenadas da pilha são usadas para obter os textos t1 e t2 do tabuleiro lógico.
        - A verificação any((t1, t2) == p or (t2, t1) == p for p in pares) lida elegantemente com a ordem, verificando se o par (t1, t2) ou (t2, t1) existe na lista pares de pares válidos.

        - A pilha.clear() é usada para resetar o estado para a próxima seleção.
3. **Desafio: Passagem de Turno e Condição de "Jogar Novamente".**
    - **Descrição do Problema:** Implementar corretamente a regra de que o jogador atual joga novamente se acertar um par, e passa a vez (turnos.rotate(-1)) apenas se errar.
    - **Solução Adotada:** Na função verificar_par(), a linha turnos.rotate(-1) está localizada especificamente dentro do bloco else (que é executado quando par_correto é False). Se par_correto é True, a rotação não ocorre, e como turnos[0] ainda se refere ao mesmo jogador, ele efetivamente joga novamente.
4. **Desafio: Layout e Responsividade da Interface com Tkinter Grid.**
    - **Descrição do Problema:** Organizar os botões/labels das cartas em uma grade que se redimensionasse razoavelmente com a janela.
    - **Solução Adotada:**
        - O uso de frame_tabuleiro como um contêiner dedicado para as cartas.
        - A configuração de frame_tabuleiro.grid_rowconfigure(i, weight=1) e frame_tabuleiro.grid_columnconfigure(j, weight=1) para as linhas e colunas do grid faz com que as células da grade se expandam para preencher o espaço disponível no frame_tabuleiro quando a janela é redimensionada.
        - O uso de sticky="nsew" ao adicionar os widgets (.grid(..., sticky="nsew")) faz com que cada widget (botão/label da carta) se expanda para preencher toda a sua célula na grade.
        - root.geometry("960x540") define um tamanho inicial e root.minsize(640, 360) um tamanho mínimo.
5. **Desafio: Evitar Chamadas Recursivas ou Bloqueio da GUI com Delays.**
    - **Descrição do Problema:** Introduzir delays (para o jogador ver as cartas) sem bloquear o loop principal do Tkinter (root.mainloop()) que processa eventos da GUI. Usar time.sleep() diretamente em uma função chamada por um evento Tkinter bloquearia a UI.
    - **Solução Adotada:** O método root.after(delay_ms, callback_function) do Tkinter foi usado. Ele agenda a callback_function para ser executada após delay_ms milissegundos, sem bloquear o loop de eventos. Isso é crucial para os delays antes de verificar_par e esconder_cartas.

A superação desses desafios resultou em um jogo funcional, interativo e educativo, demonstrando a aplicação prática dos conceitos da disciplina e habilidades de resolução de problemas no contexto do desenvolvimento com Python e Tkinter.

Conclusão Sobre o Projeto

O desenvolvimento do projeto "Jogo da Memória Visual Top", utilizando Python e a biblioteca Tkinter, representou uma aplicação prática e elucidativa dos conceitos teóricos de estruturas de dados e análise de algoritmos, fundamentais na ciência da computação. A concretização deste jogo não apenas cumpriu os requisitos acadêmicos propostos, mas também proporcionou uma imersão valiosa no ciclo de vida de um pequeno projeto de software, desde a concepção e planejamento, passando pelo desenvolvimento e testes da lógica e da interface gráfica, até a elaboração da documentação final.

A **aplicação correta e criteriosa das estruturas de dados** foi o pilar central deste trabalho.

- A **fila**, implementada com collections.deque (variável turnos), demonstrou eficácia no gerenciamento do revezamento entre os jogadores, com turnos[0] indicando o jogador atual e turnos.rotate(-1) passando a vez de forma ordenada e eficiente.
- A **pilha**, implementada com uma lista Python (variável pilha), provou ser a escolha adequada para o armazenamento temporário das coordenadas (linha, coluna) das duas cartas viradas em cada jogada, facilitando a comparação e a subsequente reversão visual das cartas (via esconder_cartas()) caso um par não fosse formado.
- A **lista** Python, em suas formas uni e bidimensional, foi extensivamente utilizada como base para a representação do tabuleiro e seus estados: pares_com_cores (definição inicial), pares (lógica de pares), cartas (para embaralhamento), tabuleiro (valores lógicos das cartas na grade), estado_tabuleiro (visibilidade 'X' ou texto), e widgets (referências aos componentes Tkinter). O acesso indexado e as list comprehensions foram cruciais.

A interação harmoniosa dessas estruturas resultou em um fluxo de jogo coerente, funcional em sua execução com interface gráfica, e eficiente em seu desempenho para o escopo definido.

A inclusão da **análise de complexidade de algoritmos (Big O)**, focada na verificação de par (verificar_par(), O(Npares)) e no embaralhamento (O(Npares)), reforçou a importância de se compreender o custo computacional. Para o tamanho fixo do jogo (6 pares), essas complexidades são excelentes e garantem a responsividade.

Além dos aspectos técnicos, o uso de ferramentas colaborativas como **Trello** e

Git/GitHub foi fundamental para o planejamento, acompanhamento e versionamento do projeto, mesmo sendo um código Python contido em um arquivo principal.

O projeto "Jogo da Memória Visual Top" não apenas atingiu seus objetivos de criar uma aplicação funcional e interativa com tema educativo, mas também proporcionou uma experiência enriquecedora em trabalho em equipe, divisão de tarefas e aplicação de boas práticas em programação Python com GUI e gestão de projetos.

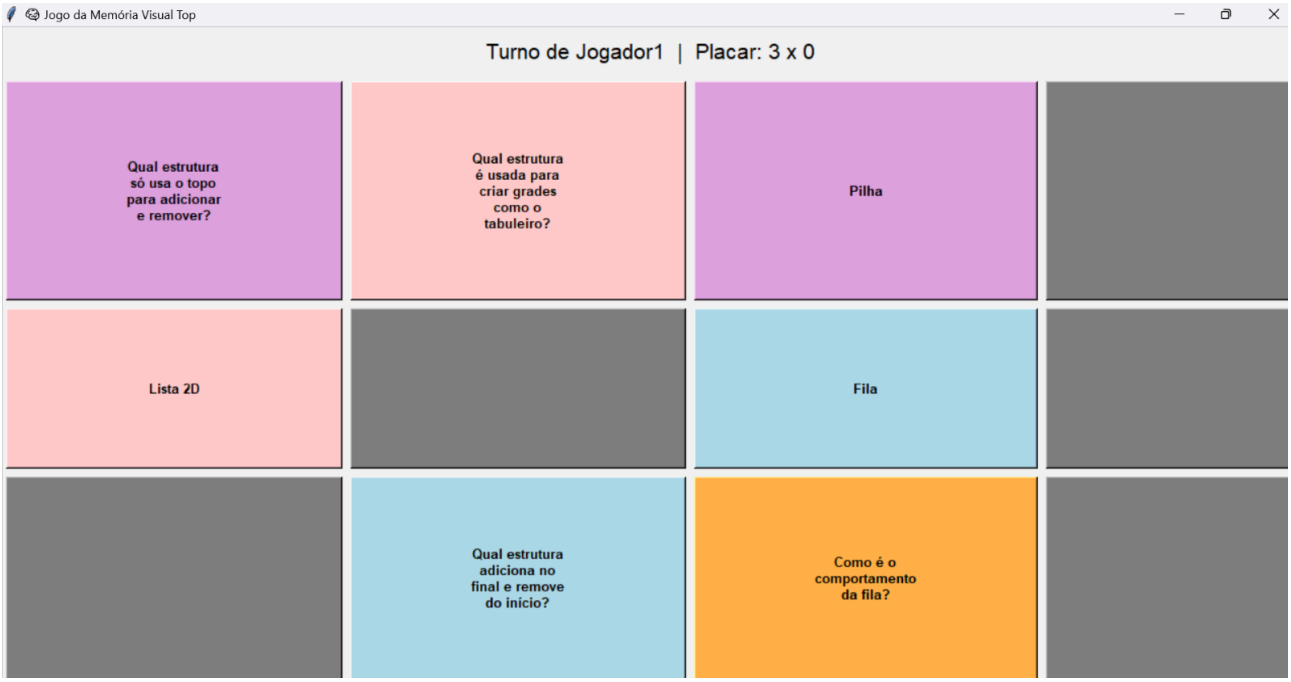
Dessa forma, este Jogo da Memória serve como um estudo de caso prático para a aplicação de estruturas de dados no desenvolvimento de jogos digitais com interface gráfica. Demonstra como conceitos de estruturas de dados se traduzem em funcionalidades concretas e como ferramentas de desenvolvimento apoiam o processo.

Por fim, este trabalho reforça a relevância do aprendizado prático aliado ao embasamento teórico. Incentiva o desenvolvimento de soluções que envolvam lógica computacional, organização de dados e colaboração – elementos indispensáveis para a formação de profissionais capacitados na área de tecnologia da informação.

Registros Visuais do Funcionamento do Jogo (Análise da Interface Gráfica - "Jogo da Memória Visual Top")

Esta seção é dedicada à apresentação e análise das capturas de tela que ilustram o "Jogo da Memória Visual Top" em plena execução. Estas imagens são fundamentais pois não apenas demonstram a interface com o usuário (GUI) desenvolvida com Tkinter, mas também fornecem evidências concretas de momentos chave da partida, como a seleção de cartas, a formação de pares, a indicação de turnos, a contagem de placar e a sinalização de fim de jogo. As capturas corroboram a funcionalidade descrita ao longo deste documento e o uso implícito das estruturas de dados para gerenciar o estado visual e lógico do jogo.

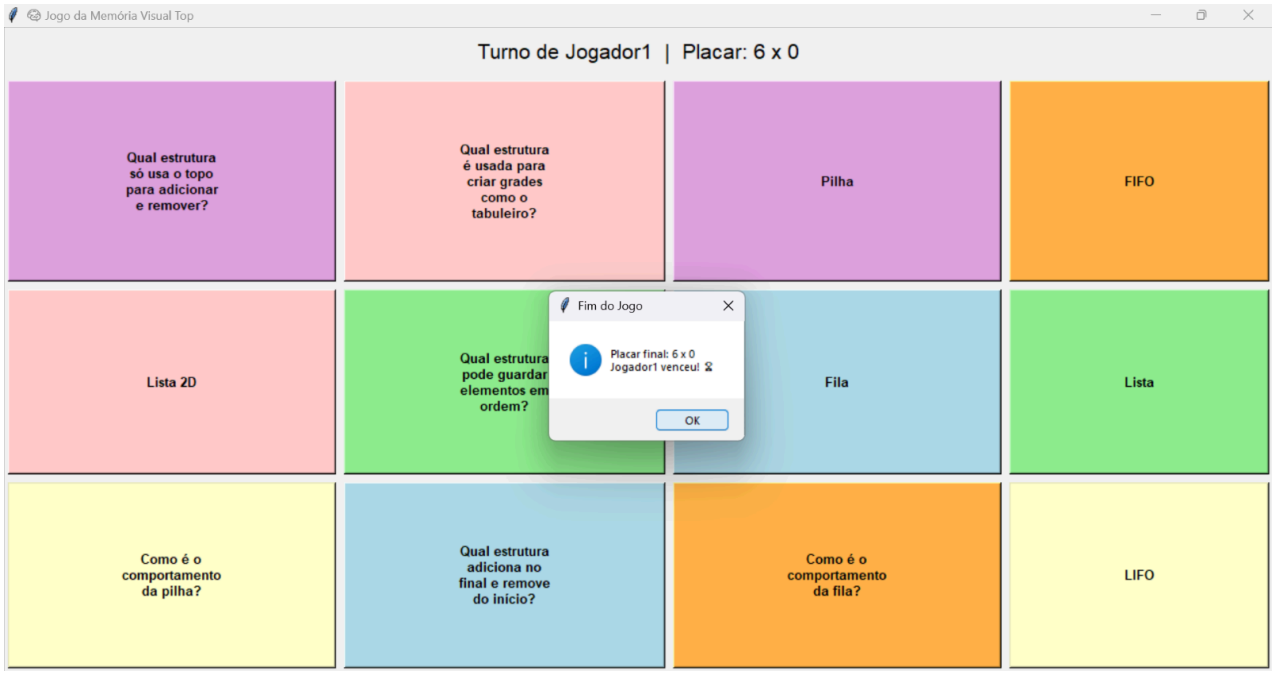
Análise da Captura de Tela 1: Início/Meio de Jogo - Turno do Jogador 1



- **Elementos Visíveis e Observações (Conectando com o Código Python):**
  - **Título da Janela:** Embora não visível na screenshot, o código define root.title("Jogo da Memória Visual Top").
  - **Indicação de Turno e Placar:** No topo, "Turno de Jogador1 | Placar: 3 x 0". Isso é gerado pela função atualizar_info() que usa turnos[0] (da deque) e pontuacao (dicionário).
  - **Tabuleiro de Cartas (Tkinter Frame com Button/Label):** O tabuleiro 3x4 é o frame_tabuleiro. As cartas ocultas são tk.Button cinzas criadas por criar_botao(). As cartas reveladas (coloridas, com texto) são tk.Label criadas

dinamicamente em virar_carta(). As cores de fundo (ex: rosa, lilás, verde claro, laranja) vêm de texto_para_cor e pares_com_cores.
    - Conteúdo das cartas: Textos como "Qual estrutura só usa o topo para adicionar e remover?" e "Pilha" são extraídos da estrutura tabuleiro (lista 2D). O wraplength=100 nos Labels e Buttons ajuda a quebrar o texto.
  - **Estado das Cartas:** Cartas cinzas correspondem a 'X' em estado_tabuleiro. Cartas reveladas têm seu texto em estado_tabuleiro. A pontuação "3 x 0" indica que 3 pares da **lista** pares foram corretamente identificados.
  - **Interação Implícita:** O estado visual é resultado de chamadas a virar_carta (que usa a **pilha** pilha para guardar as duas últimas coordenadas) e verificar_par.

Análise da Captura de Tela 2: Fim de Jogo - Vitória do Jogador 1

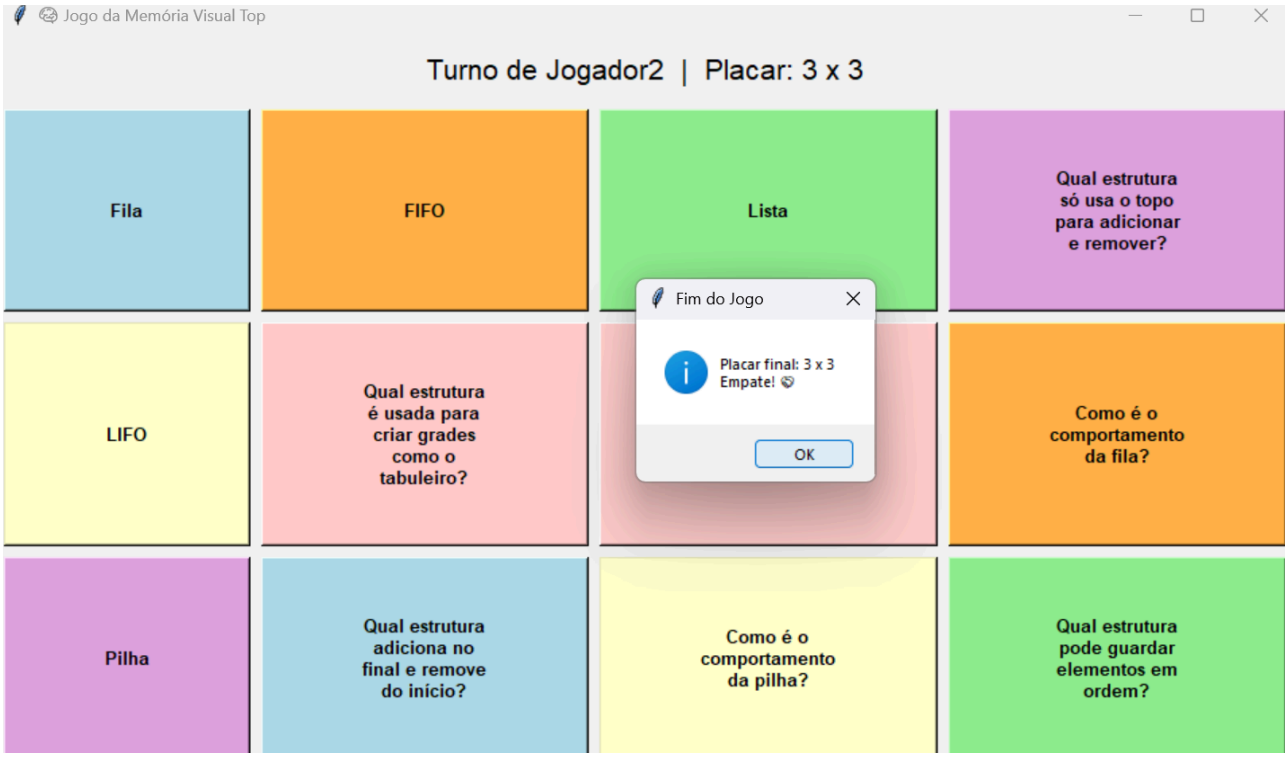


- **Elementos Visíveis e Observações (Conectando com o Código Python):**
  - **Indicação de Turno e Placar Final:** "Turno de Jogador1 | Placar: 6 x 0".
  - **Mensagem de Fim de Jogo (messagebox):** O pop-up "Fim do Jogo" com "Jogador1 venceu!" é gerado por tkinter.messagebox.showinfo() dentro da função fim_de_jogo(). A função fim_de_jogo() é chamada quando todos_revelados() (que verifica estado_tabuleiro) retorna True.
  - **Estado do Tabuleiro:** Todas as cartas estão viradas para cima (são tk.Labels coloridos), indicando que todos os 6 pares da lista pares foram encontrados.
  - **Conteúdo Educacional das Cartas:** Textos como "Como é o comportamento

da pilha?" e "LIFO" são consistentes com pares_com_cores.

Estas capturas de tela são cruciais pois fornecem uma demonstração visual da funcionalidade do jogo implementado em Python/Tkinter, validando que os conceitos de controle de turno (fila deque), gerenciamento de jogadas (lista pilha como pilha) e representação do tabuleiro (múltiplas listas e listas de listas) foram traduzidos em uma aplicação interativa e funcional.

Análise da Captura de Tela 3: Fim de Jogo - Empate



- **Elementos Visíveis e Observações (Conectando com o Código Python):**
  - **Indicação de Turno e Placar Final de Empate:** O info_label no topo da janela indica "Turno de Jogador2 | Placar: 3 x 3". Este placar igual (3 pontos para pontuacao["Jogador1"] e 3 para pontuacao["Jogador2"]) ao final da partida (quando todos_revelados() é True) leva à condição de empate.
  - **Mensagem de Fim de Jogo (Empate):** O messagebox.showinfo() dentro de fim_de_jogo() exibe o pop-up "Fim do Jogo" com a mensagem "Placar final: 3x3 Empate!". Isso confirma que a lógica de comparação de pontuações para determinar o vencedor ou o empate está funcionando corretamente.
  - **Estado Final do Tabuleiro:** Todas as 12 cartas no tabuleiro (a grade 3x4 de widgets) estão viradas para cima, exibidas como tk.Labels coloridos, indicando que todos os seis pares foram encontrados.

- **Consistência do Tema Educacional:** O conteúdo das cartas, como "Qual estrutura é usada para criar grades como o tabuleiro?" (pareada com "Lista") e "Como é o comportamento da fila?" (pareada com "FIFO"), é diretamente proveniente da estrutura pares_com_cores e tabuleiro no código Python. As cores de fundo dos tk.Labels são definidas pelo dicionário texto_para_cor.
- **Interação do Usuário:** O botão "OK" no messagebox, ao ser clicado, executaria root.quit(), encerrando a aplicação Tkinter.

Considerações Gerais sobre a Interface Gráfica (GUI) Tkinter e o Tema Escolhido:

1. **Clareza e Funcionalidade da Interface Tkinter:** As capturas de tela demonstram uma interface gráfica que, embora construída com os widgets padrão do Tkinter e visualmente simples, é **clara e funcional**. Os elementos essenciais (tabuleiro de cartas interativas, placar dinâmico, indicação de turno, mensagens de feedback modal) estão presentes e são compreensíveis. A distinção entre cartas ocultas (tk.Button) e reveladas (tk.Label) é evidente e controlada programaticamente.
2. **Feedback ao Usuário:** O jogo fornece feedback importante ao usuário através da atualização do info_label (gerenciado por atualizar_info()) e, crucialmente, das caixas de mensagem (messagebox) para sinalizar o fim da partida e o resultado. Os delays visuais com root.after() também melhoram a experiência ao permitir que o jogador veja as cartas antes de serem escondidas ou antes da verificação do par.
3. **Valor Pedagógico do Tema Implementado:** A escolha de utilizar perguntas e respostas sobre os próprios conceitos de estruturas de dados como conteúdo das cartas (definido em pares_com_cores) é uma **abordagem metalinguística e altamente relevante** para um projeto acadêmico desta natureza. Isso não apenas testa a memória dos jogadores, mas também reforça o aprendizado do conteúdo da disciplina de forma lúdica e interativa com uma aplicação Python real.
4. **Demonstração da Lógica Implementada em Python:** As diferentes telas (turno de um jogador, turno de outro, placares variados, diferentes resultados de fim de jogo) coletivamente fornecem uma evidência robusta de que a lógica de controle do jogo – incluindo a alternância de turnos (fila deque com rotate), o gerenciamento de jogadas (lista pilha como pilha), a representação do tabuleiro (listas tabuleiro, estado_tabuleiro, widgets) e a verificação de pares (iterando sobre a lista pares) – foi implementada com sucesso no código Python.
5. **Uso Efetivo de Tkinter para o Jogo:** O código demonstra um uso competente do Tkinter para criar uma aplicação de jogo interativa, incluindo:

- Estruturação da janela principal (root) e frames (frame_superior, frame_tabuleiro).
- Uso de grid para layout responsivo das cartas.
- Criação dinâmica e destruição de widgets (Button, Label) para representar os estados das cartas.
- Manipulação de eventos de clique com command=lambda....
- Uso de root.after() para temporização de ações sem bloquear a GUI.
- Uso de messagebox para comunicação com o usuário.

Em suma, os registros visuais, quando interpretados à luz do código Python fornecido, confirmam a criação de um Jogo da Memória funcional, tematicamente apropriado e que utiliza corretamente as estruturas de dados e os recursos da biblioteca Tkinter para atingir os objetivos do projeto.

Anexos Finais: Links e Papéis dos Integrantes

Repositório do Código-Fonte (GitHub)

O código-fonte completo do "Jogo da Memória Visual", desenvolvido em Python, está disponível no seguinte repositório Git hospedado na plataforma GitHub:

- **Link para o Repositório:** https://github.com/VitorDalmuth/jogo-da-memoria

Ferramenta de Gerenciamento de Tarefas (Trello)

Para a organização e o acompanhamento das tarefas durante o desenvolvimento do projeto, foi utilizada a plataforma Trello. O quadro do projeto pode ser acessado através do link:

- **Link para o Quadro do Trello:** https://trello.com/b/xdQKHpwn/a3-estrutura-de-dados

Papéis dos Integrantes

- **Emilly Ruff de Camargo:** Documentação;
- **Rafaela Martins Coelho:** Documentação e organização do Trello;
- **Vitor Luiz Dalmuth:** Definição de tema e Slide;
- **Giorgio Buka:** Slide;
- **Gabriel Jabour:** Código;
- **Guilherme Izaias Fernandes:** Documentação.