UNIVERSIDADE SALVADOR
CAMPUS TANCREDO NEVES
CURSO DE CIÊNCIA DA COMPUTAÇÃO

ANÁLISE DO ALGORITMO DE RESOLUÇÃO DA TORRE DE HANOI

USANDO COMO BASE A ESTRUTURA DE DADOS PILHA

SALVADOR
2024

Equipe:

João Amaral Lantyer ---------- RA:1271919682
Gabriel Antonio Lopes de Castro ----------RA:1272023100
Marina Fernandes Porto Leite ----------RA:1272121593
Julio Correa De Sagebin Cahú Rodrigues-----RA:1272415912

AGRADECIMENTOS

Gostaríamos de expressar nossa profunda gratidão ao professor pelo comprometimento e orientação durante o curso de Estrutura de Dados. Sua dedicação incansável em transmitir os conceitos complexos de forma acessível e inspiradora foi fundamental para o sucesso de nossa equipe.

As habilidades e o conhecimento que adquirimos sob sua tutela têm sido inestimáveis para o desenvolvimento de nosso projeto. Sua expertise e paixão pela disciplina não apenas nos incentivaram a alcançar novos patamares de compreensão, mas também nos inspiraram a perseguir a excelência em nosso trabalho.

É evidente que sua influência transcende o ambiente acadêmico e impacta diretamente em nossas trajetórias profissionais. Estamos sinceramente agradecidos por termos tido o privilégio de aprender com alguém tão respeitado e dedicado como o senhor.

Esperamos poder aplicar os conhecimentos adquiridos não apenas neste projeto, mas também em nossas carreiras futuras. Mais uma vez, agradecemos por sua orientação valiosa e por ser um exemplo inspirador para todos nós.

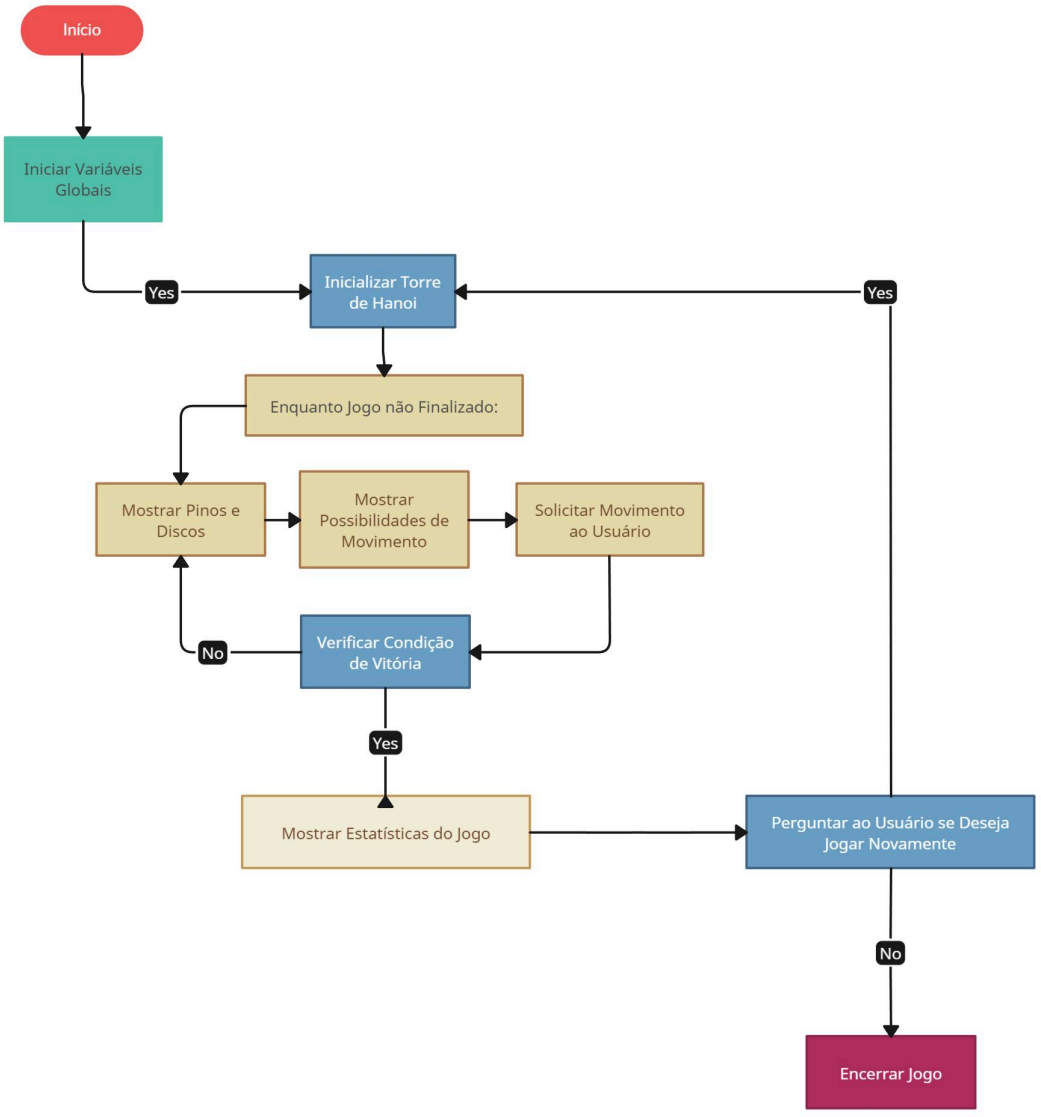
SUMÁRIO

INTRODUÇÃO AO PROJETO..... 4
Funcionamento do jogo de hanoi..... 5
LÓGICA PARA RESOLUÇÃO..... 6
2.1 Classe pilha..... 6
2.2 Funções Auxiliares..... 8
2.2.1 Função "inicializar(pino1, discos)"..... 8
2.2.2 Função "terminar(pino1, pino2, pino3)"..... 9
2.2.3 Função "movimentar(pino1, pino2, pino3)"..... 10
2.2.4 Função "mover(origem, destino)"..... 12
2.2.5 Função "mostrar_pinos(pino1, pino2, pino3)"..... 13
2.2.6 Função "mostrar_possibilidades(pino1, pino2, pino3)"..... 14
2.2.7 Função "iniciar_variaveis()"..... 15
2.2.8 Função "resolver_para_mim(pino1, pino2, pino3, n)"..... 17
2.3 Função principal "main ()":..... 18

Introdução ao projeto

A essência do projeto de Estrutura de Dados reside em destacar a vital importância da utilização e integração de múltiplas estruturas, convergindo para a implementação de um clássico desafio como a Torre de Hanói. Mais do que simplesmente fazer o código funcionar, esse projeto ilustra como a combinação estratégica de diversas estruturas é fundamental não apenas para a execução eficiente de algoritmos, mas também para a otimização do processo como um todo. Ao explorar a matéria de forma prática, buscamos não apenas alcançar a funcionalidade, mas aprimorar e aperfeiçoar qualquer estrutura de código, evidenciando sua relevância e versatilidade em cenários reais.

No prosseguimento, será apresentado um fluxograma que delineia a ideia inicial de funcionamento do jogo. Esse diagrama será de suma importância para a compreensão detalhada do projeto, fornecendo uma visão clara das etapas e interações fundamentais que compõem a mecânica do jogo.



Funcionamento do jogo de hanoi

A Torre de Hanói é um quebra-cabeça matemático composto por três pinos e discos de tamanhos distintos, empilhados em qualquer um dos pinos. O objetivo é mover todos os discos de um pino para outro, seguindo três regras simples:

1. Apenas um disco pode ser movido de cada vez.
2. Um disco só pode ser movido do topo de uma pilha para o topo de outra.
3. Um disco maior nunca pode ser colocado sobre um disco menor.

Inicialmente, os discos estão empilhados em um dos pinos, em ordem decrescente de tamanho, com o maior disco na base e o menor no topo. A resolução do jogo envolve mover os discos entre os pinos de forma a respeitar sempre a terceira regra. A complexidade do jogo aumenta exponencialmente com o número de discos, sendo que o número mínimo de movimentos necessários para resolver o quebra-cabeça com nnn discos é 2n−12^n - 12n−1.

Para implementar o algoritmo de resolução, utilizamos a estrutura de dados Pilha. Uma pilha é uma coleção ordenada de elementos que segue o princípio LIFO (Last In, First Out), ou seja, o último elemento inserido é o primeiro a ser removido. As operações básicas são:

1. Inserir: Adicionar um elemento ao topo da pilha.
2. Remover: Retirar o elemento do topo da pilha.
3. Topo: Obter o valor do elemento no topo da pilha sem removê-lo.
4. Vazio: Verificar se a pilha está vazia.
5. Tamanho: Verificar a quantidade de elementos na pilha.

No algoritmo a seguir, utilizamos um array para implementar a pilha. Cada um dos três pinos do jogo é representado como uma pilha, onde os discos são empilhados em ordem decrescente de tamanho. Assim, mover um disco entre pinos é análogo a realizar operações de retirar (desempilhar) e inserir (empilhar) nas pilhas. As regras do jogo, que proíbem colocar um disco maior sobre um disco menor, são naturalmente respeitadas pela estrutura de pilha, pois os elementos são sempre adicionados e removidos do topo.

Lógica para resolução

2.1 Classe pilha

A classe Pilha representa uma pilha de discos em uma das três torres:

```
class Pilha:
    def __init__(self, nome):
        self.itens = []
        self.nome = nome

    def empilhar(self, x):
        self.itens.append(x)

    def desempilhar(self):
        if len(self.itens) == 0:
            print("Pilha vazia")
            return -1
        return self.itens.pop()

    def tamanho(self):
        return len(self.itens)

    def topo(self):
        if len(self.itens) == 0:
            return -1
        return self.itens[-1]

    def imprimir(self):
        print("\n")
        for item in reversed(self.itens):
            print(f" " * (abs(tamanho - item)), end="")
            print(f'##' * item)

        print("[", end=" ")
        for item in self.itens:
            print(item, end=" ")
        print("]\n")
```

- Construtor __init__(self, nome): Inicializa os atributos itens como um array vazio e nome com o valor fornecido pelo usuário.

```
    def __init__(self, nome):
        self.itens = []
        self.nome = nome
```

- Método empilhar(self, x): Recebe o parâmetro x e o adiciona no topo da pilha.

```
def empilhar(self, x):
    self.itens.append(x)
```

- Método desempilhar(self): Remove e retorna o elemento do topo da pilha. Se a pilha estiver vazia, imprime "Pilha vazia" e retorna -1.

```
def desempilhar(self):
    if len(self.itens) == 0:
        print("Pilha vazia")
        return -1
    return self.itens.pop()
```

- Método tamanho(self): Retorna o número de itens na pilha.

```
def tamanho(self):
    return len(self.itens)
```

- Método topo(self): Retorna o item no topo da pilha sem removê-lo, retornando -1 se a pilha estiver vazia.

```
def topo(self):
    if len(self.itens) == 0:
        return -1
    return self.itens[-1]
```

- Método imprimir(self): Imprime os elementos da pilha de forma visual, com o topo da pilha no início. Cada elemento é representado por um número de ## correspondente ao seu valor. Também imprime todos os elementos da pilha entre colchetes no final.

```
def imprimir(self):
    print("\n")
    for item in reversed(self.itens):
        print(f" " * (abs(tamanho - item)), end="")
        print(f'##' * item)

    print("[", end=" ")
    for item in self.itens:
        print(item, end=" ")
    print("]\n")
```

2.2 Funções Auxiliares

2.2.1 Função "inicializar(pino1, discos)"

A função *inicializar* configura o estado inicial do jogo das Torres de Hanói. Ela solicita ao usuário o número de discos que deseja utilizar, valida essa entrada e, se for válida, empilha os discos no primeiro pino (pino1) em ordem decrescente.

```
def inicializar(pino1, discos):
    global tamanho
    print("\nTorre de Hanoi")
    print("A dificuldade dependerá do número de discos")
    print("Digite o número de discos que você deseja utilizar (de 2 a
50):")
    try:
        discos[0] = int(input())
    except ValueError:
        print("Digite um número inteiro!")
        return 1
    tamanho = discos[0]
    if discos[0] not in range(2, 51):
        print("\nNúmero de discos inválido!")
        return 1
    for aux in range(discos[0], 0, -1):
        pino1.empilhar(aux)
    return 2
```

Detalhamento do código:

1. Declara a variável tamanho como global para que seu valor possa ser acessado e modificado fora da função.
2. Imprime instruções sobre a Torre de Hanói e a dificuldade baseada no número de discos.
3. Solicita ao usuário que insira o número de discos desejado (de 2 a 50).
4. Tenta converter a entrada do usuário para um inteiro. Se a conversão falhar (ex.: o usuário insere uma string não numérica), um erro ValueError é capturado.
5. Se ocorrer um ValueError, imprime uma mensagem de erro e retorna 1, indicando que a inicialização falhou.

6. Verifica se o número de discos está dentro do intervalo permitido (2 a 50). Se não estiver, imprime uma mensagem de erro e retorna 1.
7. Se o número de discos for válido, armazena esse valor na variável global tamanho.
8. Utiliza um loop for para empilhar os discos no pino1 em ordem decrescente. Começa do número total de discos (discos[0]) até 1.
9. A função empilhar da classe Pilha é chamada para adicionar cada disco ao pino1.
10. Retorna 2 para indicar que a inicialização foi bem-sucedida e que o jogo está pronto para avançar para o próximo estado.

2.2.2 Função "terminar(pino1, pino2, pino3)"

A função *terminar* limpa as pilhas dos três pinos (pino1, pino2, pino3) ao final do jogo, restaurando-os ao estado inicial vazio. Esta função é chamada quando o jogo termina, seja porque o jogador resolveu o problema das Torres de Hanói ou porque o jogador optou por encerrar a sessão, ela também limpa os pinos, a função prepara o estado do jogo para um possível reinício, permitindo que o jogador comece um novo jogo sem resíduos do jogo anterior.

```
def terminar(pino1, pino2, pino3):
    pino1.itens.clear()
    pino2.itens.clear()
    pino3.itens.clear()
```

Detalhamento do código:

1. pino1.itens.clear(): Remove todos os elementos da lista itens do pino1.
2. pino2.itens.clear(): Remove todos os elementos da lista itens do pino2.
3. pino3.itens.clear(): Remove todos os elementos da lista itens do pino3.

2.2.3 Função "movimentar(pino1, pino2, pino3)"

Dentro do ciclo do jogo, a função *movimentar* é chamada quando o estado do jogo requer um movimento do usuário, dependendo do retorno da função movimentar, o estado do jogo é atualizado para continuar solicitando movimentos, resolver automaticamente, ou finalizar o jogo. Ela irá processar a entrada do usuário para realizar um movimento no jogo das Torres de Hanói ou para ativar a solução automática. Ela interpreta o comando do usuário, valida o movimento, e, se válido, executa o movimento e atualiza o estado do jogo. A função também lida com entradas inválidas e garante que o jogo não quebre devido a movimentos incorretos.

```
def movimentar(pino1, pino2, pino3):
    global primeiro_turno
    if primeiro_turno:
        movimento = input("Digite qual destes movimentos você deseja
realizar (ou digite 'RESOLVA PARA MIM' para resolver automaticamente):
").upper()
    else:
        movimento = input("Digite qual destes movimentos você deseja
realizar: ").upper()

    if primeiro_turno and movimento == "RESOLVA PARA MIM":
        resolver_para_mim(pino1, pino2, pino3, tamanho)
        primeiro_turno = False
        return 6

    if movimento[0] not in ["A", "B", "C"] or movimento[1] not in ["A",
"B", "C"]:
        print("\n MOVIMENTO INVÁLIDO! \n")
        return 2

    origem = None
    destino = None

    if movimento[0] == "A":
        origem = pino1
    elif movimento[0] == "B":
        origem = pino2
    elif movimento[0] == "C":
        origem = pino3

    if movimento[1] == "A":
        destino = pino1
```

```
    elif movimento[1] == "B":
        destino = pino2
    elif movimento[1] == "C":
        destino = pino3

    if mover(origem, destino) == 0:
        return 4
    global contador_movimentos
    contador_movimentos += 1
    movimentos.append(f"{movimento[0]}{movimento[1]}")
    primeiro_turno = False
    return 5
```

Detalhamento da função:

1. A variável primeiro_turno é usada para verificar se é a primeira jogada do usuário. Dependendo disso, a mensagem de solicitação de entrada pode variar.
2. Se for o primeiro turno, o usuário pode optar por resolver automaticamente digitando "RESOLVA PARA MIM". Em todos os outros turnos, o usuário deve digitar os movimentos manualmente. O movimento é transformado em letras maiúsculas para garantir consistência.
3. Se o usuário digitar "RESOLVA PARA MIM" no primeiro turno, a função resolver_para_mim é chamada para resolver automaticamente o problema das Torres de Hanói. Após chamar resolver_para_mim, primeiro_turno é definido como False e a função retorna 6 para a função principal, indicando que o jogo deve avançar para o estado 6.
4. O movimento deve ser composto por duas letras (pinos de origem e destino) que devem ser "A", "B" ou "C". Se a entrada não for válida, imprime uma mensagem de erro e retorna 2, indicando que o usuário deve tentar novamente.
5. A função traduz as letras dos movimentos (movimento[0] e movimento[1]) para as pilhas correspondentes (pino1, pino2, pino3).
6. A função mover é chamada para realizar o movimento entre os pinos determinados. Se mover retornar 0, significa que o movimento é inválido (tentativa de colocar um disco maior sobre um menor) e a função retorna 4 para indicar que o jogo deve continuar aguardando uma entrada válida.

7. Se o movimento for válido, o contador de movimentos (contador_movimentos) é incrementado. O movimento é adicionado à lista de movimentos realizados (movimentos). primeiro_turno é definido como False e a função retorna 5, indicando que o movimento foi bem-sucedido e que o jogo deve continuar.

2.2.4 Função "mover(origem, destino)"

A função *mover* tem como objetivo mover um disco do topo da pilha de origem para o topo da pilha de destino, seguindo as regras das Torres de Hanói, que proíbem a colocação de um disco maior sobre um disco menor. Ela verifica se o movimento é válido antes de realizá-lo. Se o movimento for inválido, o usuário recebe um feedback imediato através de uma mensagem de erro, ajudando a reforçar as regras do jogo. A função *mover* é chamada dentro da função *movimentar* para verificar e executar o movimento. Se o movimento for inválido (*mover* retorna 0), *movimentar* retorna 4, indicando que o jogo deve continuar aguardando uma entrada válida. Se o movimento for válido, o contador de movimentos é incrementado, o movimento é registrado e o jogo continua.

```
def mover(origem, destino):
    if origem.tamanho() > 0 and (destino.tamanho() <= 0 or origem.topo()
< destino.topo()):
        destino.empilhar(origem.desempilhar())
        return 1
    else:
        print("\nNão é possível realizar este movimento, você deve
colocar pinos menores acima de maiores.\n")
        return 0
```

Detalhamento da função:

1. A função primeiro verifica se a pilha de origem (origem) tem discos (origem.tamanho() > 0). Em seguida, verifica se a pilha de destino (destino) está vazia (destino.tamanho() <= 0) ou se o disco no topo da pilha de origem é menor que o disco no topo da pilha de destino (origem.topo() < destino.topo()).

2. Se ambas as condições acima forem verdadeiras, o movimento é considerado válido. O disco do topo da pilha de origem é removido (origem.desempilhar()) e empilhado na pilha de destino (destino.empilhar()). A função retorna 1, indicando que o movimento foi realizado com sucesso.
3. Se qualquer uma das condições de validade do movimento falhar, o movimento é considerado inválido. Uma mensagem de erro é impressa para informar ao usuário que não é possível realizar o movimento, pois discos menores devem ser colocados sobre discos maiores. A função retorna 0, indicando que o movimento não foi realizado.

2.2.5 Função "mostrar_pinos(pino1, pino2, pino3)"

A função *mostrar_pinos* exibe o estado atual de cada uma das três pilhas (ou "pinos") utilizadas no jogo das Torres de Hanói. Isso proporciona uma visualização clara e imediata para o jogador sobre a posição e a ordem dos discos em cada pilha. A função *mostrar_pinos* é chamada em vários pontos do loop principal do jogo para atualizar a visualização do estado das pilhas. Após cada ação relevante (como a inicialização dos pinos, a movimentação de discos ou a finalização do jogo), a função *mostrar_pinos* é chamada para atualizar e exibir o estado atual das pilhas.

```
def mostrar_pinos(pino1, pino2, pino3):
    print("\nA: ", end="")
    pino1.imprimir()
    print("B: ", end="")
    pino2.imprimir()
    print("C: ", end="")
    pino3.imprimir()
```

Detalhamento da função:

1. Impressão do Estado do Pino 1:
    a. *print("\nA: ", end="")*: Imprime a identificação do primeiro pino (pino1) com um rótulo "A: ".

    b. *pino1.imprimir()*: Chama o método *imprimir* do objeto pino1 para exibir o estado atual dos discos no primeiro pino.

2. Impressão do Estado do Pino 2:
    a. *print("B: ", end="")*: Imprime a identificação do segundo pino (pino2) com um rótulo "B: ".
    b. *pino2.imprimir()*: Chama o método *imprimir* do objeto pino2 para exibir o estado atual dos discos no segundo pino.
3. Impressão do Estado do Pino 3:
    a. *print("C: ", end="")*: Imprime a identificação do terceiro pino (pino3) com um rótulo "C: ".
    b. *pino3.imprimir()*: Chama o método *imprimir* do objeto pino3 para exibir o estado atual dos discos no terceiro pino.

2.2.6 Função "mostrar_possibilidades(pino1, pino2, pino3)"

A função *mostrar_possibilidades* lista e exibe todos os movimentos válidos possíveis no momento atual do jogo das Torres de Hanói. Isso ajuda o jogador a saber quais são as ações permitidas de acordo com o estado atual das pilhas. Essa função é chamada no loop principal do jogo para exibir os movimentos válidos após cada ação do jogador

```
def mostrar_possibilidades(pino1, pino2, pino3):
    print("Movimentos válidos: ", end="")
    if (pino1.topo() < pino2.topo() or pino2.tamanho() == 0) and
pino1.tamanho() > 0:
        print("AB ", end="")
    if (pino1.topo() < pino3.topo() or pino3.tamanho() == 0) and
pino1.tamanho() > 0:
        print("AC ", end="")
    if (pino2.topo() < pino1.topo() or pino1.tamanho() == 0) and
pino2.tamanho() > 0:
        print("BA ", end="")
    if (pino2.topo() < pino3.topo() or pino3.tamanho() == 0) and
pino2.tamanho() > 0:
        print("BC ", end="")
    if (pino3.topo() < pino1.topo() or pino1.tamanho() == 0) and
pino3.tamanho() > 0:
        print("CA ", end="")
    if (pino3.topo() < pino2.topo() or pino2.tamanho() == 0) and
pino3.tamanho() > 0:
```

```
        print("CB ", end="")
    print()
```

Detalhamento da função:

1. A função começa imprimindo "Movimentos válidos: " para indicar ao jogador que a lista de movimentos possíveis será mostrada.
2. Para cada par de pinos (pino1, pino2, pino3), a função verifica se o movimento é válido. A verificação é feita usando as seguintes condições:
    a. Se o topo do pino1 é menor que o topo do pino2 ou se o pino2 está vazio (pino2.tamanho() == 0), e se pino1 não está vazio (pino1.tamanho() > 0), então o movimento de pino1 para pino2 é válido, e a função imprime "AB ".
    b. Se o topo do pino1 é menor que o topo do pino3 ou se o pino3 está vazio (pino3.tamanho() == 0), e se pino1 não está vazio (pino1.tamanho() > 0), então o movimento de pino1 para pino3 é válido, e a função imprime "AC ".
    c. Se o topo do pino2 é menor que o topo do pino1 ou se o pino1 está vazio (pino1.tamanho() == 0), e se pino2 não está vazio (pino2.tamanho() > 0), então o movimento de pino2 para pino1 é válido, e a função imprime "BA ".
    d. Se o topo do pino2 é menor que o topo do pino3 ou se o pino3 está vazio (pino3.tamanho() == 0), e se pino2 não está vazio (pino2.tamanho() > 0), então o movimento de pino2 para pino3 é válido, e a função imprime "BC ".
    e. Se o topo do pino3 é menor que o topo do pino1 ou se o pino1 está vazio (pino1.tamanho() == 0), e se pino3 não está vazio (pino3.tamanho() > 0), então o movimento de pino3 para pino1 é válido, e a função imprime "CA ".
    f. Se o topo do pino3 é menor que o topo do pino2 ou se o pino2 está vazio (pino2.tamanho() == 0), e se pino3 não está vazio (pino3.tamanho() > 0), então o movimento de pino3 para pino2 é válido, e a função imprime "CB ".

3. A função termina imprimindo um espaço em branco para separar a lista de movimentos válidos das outras mensagens que possam seguir.

2.2.7 Função "iniciar_variaveis()"

A função *iniciar_variaveis* tem o objetivo de inicializar e resetar variáveis globais que são essenciais para o funcionamento do jogo das Torres de Hanói. A função é chamada antes da inicialização do estado do jogo e da configuração das pilhas. Isso garante que todos os contadores e listas estejam limpos antes de qualquer lógica de jogo ser executada.

```
def iniciar_variaveis():
    global movimentos
    global contador_movimentos
    global primeiro_turno
    movimentos = []
    contador_movimentos = 0
    primeiro_turno = True
```

Detalhamento da função:

1. Declarando Variáveis Globais:
    a. *global movimentos*: Declara a variável movimentos como global, indicando que ela será utilizada em todo o programa.
    b. *global contador_movimentos*: Declara a variável contador_movimentos como global.
    c. *global primeiro_turno*: Declara a variável primeiro_turno como global.
2. Inicializando a Lista de Movimentos:
    a. *movimentos = [ ]*: Inicializa movimentos como uma lista vazia. Esta lista será utilizada para armazenar todos os movimentos realizados durante o jogo.
3. Inicializando o Contador de Movimentos:

a. *contador_movimentos = 0*: Inicializa contador_movimentos com o valor zero. Este contador será incrementado a cada movimento válido realizado no jogo.

4. Inicializando o Estado do Primeiro Turno:

   a. *primeiro_turno = True*: Inicializa primeiro_turno com o valor True. Esta variável é usada para verificar se é o primeiro turno do jogo, permitindo o oferecimento de uma opção especial para resolver o jogo automaticamente no primeiro turno.

2.2.8 Função "resolver_para_mim(pino1, pino2, pino3, n)"

A função *resolver_para_mim* implementa a solução automática para o problema das Torres de Hanói. Essa função utiliza uma abordagem recursiva para mover os discos de uma pilha (origem) para outra (destino), utilizando uma pilha auxiliar. A função principal é *resolver*, que é definida dentro de *resolver_para_mim*. Esta função é chamada quando o jogador escolhe a opção "RESOLVA PARA MIM". Ela mostra cada movimento passo a passo. A função *resolver_para_mim* é invocada no contexto do loop principal do jogo, quando o jogador escolhe resolver automaticamente.

```
def resolver_para_mim(pino1, pino2, pino3, n):

    def resolver(n, origem, destino, auxiliar):
        if n == 0:
            return
        resolver(n - 1, origem, auxiliar, destino)
        destino.empilhar(origem.desempilhar())
        global contador_movimentos
        contador_movimentos += 1
        movimentos.append((origem.nome, destino.nome))
        mostrar_pinos(pino1, pino2, pino3)
        resolver(n - 1, auxiliar, destino, origem)

    resolver(n, pino1, pino3, pino2)
```

Detalhamento da função:

1. Definição da Função *resolver_para_mim*:
    a. Esta função serve como um contêiner para a função recursiva *resolver* e é responsável por iniciar o processo de resolução automática das Torres de Hanói.
2. Definição da Função Recursiva *resolver*:
    a. *resolver* é uma função interna dentro de *resolver_para_mim*. Ela utiliza a abordagem clássica de recursão para resolver o problema das Torres de Hanói.
3. Condição Base da Recursão:
    a. *if n == 0*: return: Se o número de discos (n) for zero, a função retorna imediatamente, pois não há discos para mover.
4. Movendo Discos Recursivamente:
    a. *resolver(n - 1, origem, auxiliar, destino):* Primeiro, move os n-1 discos da pilha origem para a pilha auxiliar, utilizando a pilha destino como auxiliar.
    b. *destino.empilhar(origem.desempilhar())*: Move o disco restante da pilha origem para a pilha destino.
    c. *contador_movimentos += 1*: Incrementa o contador de movimentos global a cada movimento realizado.
    d. *movimentos.append((origem.nome, destino.nome))*: Adiciona o movimento realizado à lista global de movimentos.
    e. *mostrar_pinos(pino1, pino2, pino3)*: Mostra o estado atual das pilhas após cada movimento.
    f. *resolver(n - 1, auxiliar, destino, origem)*: Finalmente, move os n-1 discos da pilha auxiliar para a pilha destino, utilizando a pilha origem como auxiliar.

5. Chamada Inicial da Função Recursiva:

    a. *resolver(n, pino1, pino3, pino2)*: Inicia o processo de resolução movendo n discos da pilha pino1 para a pilha pino3, utilizando a pilha pino2 como auxiliar.

2.3 Função principal "main ()":

Esta função é a principal do programa e controla o fluxo geral do jogo das Torres de Hanói, incluindo a interação com o jogador, a resolução automática e a finalização do jogo.

```
def main():
    while True:
        iniciar_variaveis()
        discos = [0]
        A = Pilha("A")
        B = Pilha("B")
        C = Pilha("C")
        estado = inicializar(A, discos)

        while estado != 1000:
            if estado == 1:
                estado = inicializar(A, discos)
            elif estado == 2:
                mostrar_pinos(A, B, C)
                estado = 3
            elif estado == 3:
                mostrar_possibilidades(A, B, C)
                estado = 4
            elif estado == 4:
                estado = movimentar(A, B, C)
            elif estado == 5:
                if C.tamanho() == discos[0]:
                    estado = 6
                else:
                    estado = 2
            elif estado == 6:
                mostrar_pinos(A, B, C)
                terminar(A, B, C)
                estado = 1000

        print("Parabéns!")
        print("Jogo finalizado!")
```

```
        movimentos_formatados = " ".join([f"{origem}{destino}" for
origem, destino in movimentos])
        print(f"Movimentos realizados em ordem: {movimentos_formatados}")
        print(f"Total de movimentos: {contador_movimentos}\n")

        escolha = input("Deseja jogar novamente? Digite SIM para
reiniciar ou qualquer outra palavra para encerrar: ").upper()
        if escolha != "SIM":
            break
```

1. Estrutura de Repetição Principal:

Loop While: O programa principal é executado dentro de um loop while True, que continua indefinidamente até que o jogador decida encerrar o jogo.

2. Inicialização do Jogo:
    a. *iniciar_variaveis()*: Chama a função para inicializar as variáveis globais necessárias para o jogo.
    b. *discos = [0]*: Cria uma lista para armazenar o número de discos a serem usados no jogo.
    c. *Pilhas A, B e C*: Cria três instâncias da classe Pilha representando as torres do jogo.

3. Lógica do Jogo
    a. *estado = inicializar(A, discos)*: Inicializa o estado do jogo, retornando um código que indica o próximo passo a ser tomado.
    b. Loop while Interno: Controla o fluxo do jogo enquanto o estado não for igual a 1000 (indicando o fim do jogo).
        i. Condições if-elif: Verifica o estado atual do jogo e toma ações correspondentes com base nesse estado.
        ii. Chamadas de Funções: Chama as funções inicializar, *mostrar_pinos*, *mostrar_possibilidades*, *movimentar*, *mostrar_pinos* e terminar conforme necessário para interagir com o jogador e avançar o jogo.
4. Finalização do Jogo:
    a. *print("Parabéns!"):* Informa ao jogador que o jogo foi concluído com sucesso.

b. Mostra Movimentos: Formata e exibe os movimentos realizados durante o jogo.
c. Pergunta ao Jogador: Pergunta se o jogador deseja jogar novamente.
d. Condição de Saída: Se o jogador não deseja jogar novamente, o loop principal é interrompido e o programa é encerrado.

Concluindo, o jogo utiliza um sistema de estados para controlar o fluxo, o que facilita a organização e compreensão do código. Ele também oferece opções ao jogador e fornece feedback durante todo o jogo. Por fim, ressaltamos a utilização de funções definidas anteriormente, como *inicializar*, *mostrar_pinos*, *movimentar*, etc., para modularizar o código e torná-lo mais legível e fácil de manter.
Essa função main é a espinha dorsal do programa, coordenando todas as interações entre o jogador e o jogo, garantindo uma experiência interativa e envolvente durante a resolução das Torres de Hanói.