Relatório: Trabalho 2 de Projeto e Análise de Algoritmos

Alunos: Gustavo Teixeira Magalhães, Davi Arantes.

1. Lógica do Problema e Desenvolvimento Teórico

O paradigma da indução é uma forma de resolver algoritmos muito comum, além de ser um modo fácil e rápido de resolver um problema. A indução precisa de duas coisas básicas:

1. Um caso base.
2. Um passo indutivo, que resolva instâncias menores do problema.

Para resolver o problema das torres de Hanoi por meio da indução recursiva, utilizamos do mesmo modelo:

1. Caso base: Se existir somente um disco no pino de origem, mova-o para o pino de destino, e a torre estará completa.
2. Passo indutivo:
    a. Mova os pinos menores do pino de origem para o pino auxiliar, utilizando o pino de destino como auxiliar.
    b. Mova o pino maior do pino de origem para o pino de destino.
    c. Caso não seja possível realizar o passo anterior, mova os discos do pino auxiliar até o pino de destino utilizando o pino de origem como auxiliar.
    d. Repita o processo até chegar no caso base e resolver o problema.

A nível de código, resolver esse problema com indução recursiva é muito simples: crie uma função que tem como parâmetro o número de discos a serem empilhados e os nomes das torres: origem, destino e auxiliar.Na função verifique se o problema está no caso base, se estiver, mostre que moveu os discos do pino de origem pro destino, e retorne a função. Caso contrário, chame novamente a função, enviando os discos restantes, e trocando o pino auxiliar com o destino, de modo a satisfazer o passo “b”. Mostre na tela que os n-1 pinos da origem foram transferidos para o pino auxiliar. Por fim, chame novamente a função, mandando os n-1 pinos, trocando o pino auxiliar com o pino de origem, de modo a reorganizar os pinos e satisfazer a condição “c”.

2. Estruturação do Trabalho:

O programa foi desenvolvido sobre uma estrutura de orientação a objetos, na qual existe uma classe principal:

A classe Hanoi, que possui toda a lógica envolvida na troca de discos pelas torres, onde existem dois métodos separados: Um para apenas a exibição das transições feitas. E outro para a exibição gráfica das torres sendo empilhadas e desempenhadas.

Todos os métodos dessas classes são estáticos, o que é evidenciado pela keyword @staticmethod do python.

Além das classes principais, o projeto tem um arquivo main.py que serve como um menu e chama algumas funções. Sendo elas:

comoFunciona() - que tem como papel informar ao usuário de como é feito o funcionamento do jogo e do algoritmo usado para a resolução do mesmo.

hanoi() - é o método dentro da classe Hanoi responsável por resolver e exibir todas as transições feitas pelo algoritmo.

hanoiPilha() - é outro método responsável pela resolução das transições feitas pelo algoritmo. O diferencial entre elas está apenas no modo de exibição. Onde a *hanoiPilha()* exibe as transições uma por uma no formato das torres formadas.

criaPilha() - é a função responsável pela criação das pilhas usadas apenas para a exibição do método *hanoiPilha()*.

3. A Classe Hanoi:

A classe Hanoi é responsável por toda a parte de lógica e exibição das transições feitas. Ela é composta por dois métodos de exibição, que são:

a. *hanoi():*

```
def hanoi(n, origem = 'A', destino = 'C', auxiliar = 'B'):
    if n == 1:
        print("Move de {} pra {}".format(origem, destino))
    else:
        Hanoi.hanoi(n-1, origem, auxiliar, destino)
        print("Move de {} pra {}".format(origem, destino))
        Hanoi.hanoi(n-1, auxiliar, destino, origem)
```

Esse é o método principal pela resolução e exibição final do resultado. Ela recebe como parâmetro um número inteiro de discos (n), e três strings com o nome de cada uma das torres. Primeiro é feita a verificação se a chamada atual é um caso base, onde só existe um disco na origem. Caso isso seja verdade ele printa a transição e retorna as torres feitas. Caso não seja verdadeiro o ele chama a recursão e passa como parâmetro o (número de discos) - 1, a mesma origem da chamada anterior mas ele troca a torre auxiliar com a torre de destino para usar a torre auxiliar como destino com intuito de se chegar ao caso base. Quando o caso base ocorre ele passa o maior disco localizado na origem para o destino. Quando isso acontece, ele troca o auxiliar com a origem e mantém o destino. E repete esse procedimento até a resolução da torre.

b. *hanoiPilha():*

```
def hanoiPilha(n, origem, destino, auxiliar, t1 = 'A', t2 = 'C', t3 = 'B'):
    if n == 1:
        valor = origem.pop()
        destino.append(valor)
        exibeTorres(origem, auxiliar, destino, t1, t2, t3)
    else:
        Hanoi.hanoiPilha(n-1, origem, auxiliar, destino)
        valor = origem.pop()
        destino.append(valor)
        exibeTorres(origem, auxiliar, destino, t1, t2, t3)
        Hanoi.hanoiPilha(n-1, auxiliar, destino, origem)
```

Esse método tem a lógica de funcionamento idêntica ao anterior porém, com a alteração no modo de exibição. A exibição aqui é feita pela função *exibeTorres()* que usa de 3 pilhas recebidas como

parâmetro onde inicialmente apenas a pilha mais a esquerda está preenchida com números ordenados de forma decrescente. Quando é feita uma transição o número é retirado do topo da pilha origem através da função *origem.pop()* e é colocado no topo da pilha destino através da função *destino.append()* após isso, a função *exibeTorres()* é chamada e o processo se repete até a conclusão de todas as transições.

4. Função *exibeTorres():*

Função responsável pela exibição das transições sendo feitas pelas torres.

```
def exibeTorres(A, B, C, t1, t2, t3):
    cls()
    tam = max(len(A), len(B), len(C))
    for i in range(tam - 1, -1, -1):
        for torre in [A, B, C]:
            if len(torre) <= i:
                print('\t\t', end='')
            else:
                print(f'\t   {torre[i]}  ', end='')
        print()

    print('\t |', t1, '| \t |', t2, '| \t |', t3, '|')
    time.sleep(1)
```

Essa é a função responsável pela exibição das 3 pilhas passadas como parâmetro todas vez que uma transição é feita no método *hanoiPilha()*. Ela recebe como parâmetro as pilhas no momento atual da chamada, e o nome dado a cada torre. Ela usa o tamanho da maior pilha como tamanho máximo para representar o design das três torres . O restante é apenas a repetição feita para a representação com os prints necessários. Ao final ele printa o nome das torres e tem um pequeno atraso de tempo para possibilitar a leitura.

5. Funções auxiliares

a. *criaPilha():*

```
def criaPilha(tamanho):
    for i in range(tamanho):
        A.append(tamanho)
        tamanho = tamanho-1
    return A
```

Função responsável por criar a pilha ordenada de maneira decrescente com apenas números inteiros do número escolhido de discos a 1.

b. cls():

```
def cls():
    os.system('cls' if os.name == 'nt' else 'clear')
```

Função que chama a biblioteca *os* usada para limpar o console. Ela é usada apenas como uma comodidade onde para limpar o console chamamos apenas o nome dela *cls()* ao invés de chamarmos toda a função *os.system()*.

6. Conclusão e Desafios Encontrados

Definitivamente, o trabalho das torres de Hanói foi o mais tranquilo de se implementar até o momento. A dificuldade vem na forma da abstração do problema: identificar e traduzir em código a teoria por trás do processo é uma tarefa que a princípio parecia nebulosa, mas a partir do momento que se entende o funcionamento do passo recursivo, a implementação fica bem mais simples.