Permutação de Árvore Genealógica

Sobre o Projeto

O projeto tem como objetivo calcular as permutações de uma árvore genealógica onde só é permitido colocar nome e idade de cada membro da família e, de acordo com os casos de negócio, o programa irá calcular todas as possibilidades.

usando...

- python
- sem bibliotecas de cálculo
- permutação e combinação
- sem e com responsividade

Casos de Negócio

Para que o programa fique o mais perto da realidade possível, criamos alguns casos de negócio. Claro, nem todas as famílias seguem esse padrão, mas foi nossa solução para que a árvore fique mais coesa e mais realista.

1. Uma pessoa mais nova não pode ser genitora de uma pessoa mais velha e se a diferença de idades delas forem maior de 15 anos, elas tem relação de filho e pai.

2. Se a diferença de idade entre duas pessoas for menos de 15 anos ou igual a 15 anos, elas devem ser consideradas irmãos ou um casal;

3. Se a diferença de idade entre duas pessoas for mais de 50 anos e menos de 80, elas devem ser consideradas avô e neto;

4. Se a diferença de idade entre duas pessoas for mais de 80 anos, elas devem ser consideradas bisavô e bisneto.

Tipos de mecanismos utilizados

Recursividade

1. A recursividade é um tipo de mecanismo de programação onde um método/função “se chama” nele mesmo, onde se tem um empilhamento de memória de diversas ações semelhantes para cada passagem do código.

2. O mecanismo de recursividade em nosso projeto está presente na função que gera permutações válidas e que chama ele mesmo até que todas as pessoas estejam conectadas.

Iteração

1. A iteração é um mecanismo de programação onde várias instruções e métodos/funções são repetidas varias vezes, ate que seja atingido algum resultado estipulado. Os mais comuns para representarem a iteração são (for) e (while).

2. O mecanismo de iteração em nosso projeto foi utilizado na formação e permutação das relações dos familiares que comporão a arvore genealogica

Análise Sintática do Código Recursivo

Função *definir_relacoes_possiveis*

```
# Função para definir as relações possíveis entre duas pessoas com base nas idades e regras
def definir_relacoes_possiveis(pessoa1, pessoa2):
    nome1, idade1 = pessoa1
    nome2, idade2 = pessoa2
    relacoes = []

    # Pai/mãe e filho: diferença de idade entre 20 e 50 anos
    if idade1 > idade2 and 20 <= (idade1 - idade2) <= 50:
        relacoes.append('pai/mãe')
    elif idade2 > idade1 and 20 <= (idade2 - idade1) <= 50:
        relacoes.append('filho')

    # Irmãos: diferença de idade de até 15 anos e ambos menores de 50 anos
    if abs(idade1 - idade2) <= 15 and idade1 < 50 and idade2 < 50:
        relacoes.append('irmão/irmã')

    # Casal: diferença de idade de até 15 anos e ambos maiores ou iguais a 18 anos
    if abs(idade1 - idade2) <= 15 and idade1 >= 18 and idade2 >= 18:
        relacoes.append('casal')

    # Avô/avó e neto: diferença de idade de pelo menos 50 anos
    if idade1 > idade2 and (idade1 - idade2) >= 50:
        relacoes.append('avô/avó')
    elif idade2 > idade1 and (idade2 - idade1) >= 50:
        relacoes.append('neto')
```

```
# Bisavô/bisavó e bisneto: diferença de idade de pelo menos 80 anos
if idade1 > idade2 and (idade1 - idade2) >= 80:
    relacoes.append('bisavô/bisavó')
elif idade2 > idade1 and (idade2 - idade1) >= 80:
    relacoes.append('bisneto')

return relacoes
```

Função *gerar_combinacoes*

```
# Função para gerar todas as combinações possíveis de relações entre membros
def gerar_combinacoes(membros):
    combinacoes_possiveis = []

    # Percorrer todos os pares de membros para determinar suas relações
    for i in range(len(membros)):
        for j in range(i + 1, len(membros)):
            pessoa1 = membros[i]
            pessoa2 = membros[j]

            # Procurar as possíveis relações entre pessoa1 e pessoa2
            relacoes = definir_relacoes_possiveis(pessoa1, pessoa2)

            for relacao in relacoes:
                combinacoes_possiveis.append([pessoa1[0], pessoa2[0], relacao])

    return combinacoes_possiveis
```

Análise Sintática do Código Recursivo

Função *arvore_genealogica*

```python
# Função principal para receber a entrada e gerar a árvore genealógica
def arvore_genealogica():
    membros = []
    print("Digite o nome e a idade dos membros da família. Digite 'sair' para finalizar.")

    # Receber os membros da família do usuário
    while True:
        nome = input("Nome: ").strip()
        if nome.lower() == 'sair':
            break
        try:
            idade = int(input(f"Idade de {nome}: ").strip())
            if idade <= 0:
                print("A idade deve ser um número positivo.")
                continue
            membros.append((nome, idade))
        except ValueError:
            print("Por favor, insira um número válido para a idade.")

    # Verificar se há pelo menos duas pessoas
    if len(membros) < 2:
        print("É necessário pelo menos duas pessoas para gerar a árvore genealógica.")
        return

    # Gerar todas as combinações possíveis
    combinacoes = gerar_combinacoes(membros)

    # Gerar todas as permutações possíveis que conectem todos os membros
    permutacoes = gerar_permutacoes_validas(membros, combinacoes)

    # Exibir as permutações possíveis
    if permutacoes:
        print("\nPermutações possíveis da árvore genealógica:")
        for idx, permutacao in enumerate(permutacoes):
            print(f"Permutação {idx + 1}:")
            for relacao in permutacao:
                print(f"  {relacao[0]} é {relacao[2]} de {relacao[1]}")
            print()
    else:
        print("\nNenhuma permutação possível foi encontrada.")
```

Função *gerar_permutações_validas*

```python
# Função para gerar permutações que conectem todas as pessoas, sem sobreposição de relações
def gerar_permutacoes_validas(membros, combinacoes):
    permutacoes_possiveis = []

    # Função recursiva para gerar todas as permutações
    def gerar(permutacao_atual, pessoas_na_permutacao):
        # Se todas as pessoas foram conectadas, adicionar permutação à lista
        if len(pessoas_na_permutacao) == len(membros):
            permutacoes_possiveis.append(list(permutacao_atual))
            return

        # Verificar todas as combinações para adicionar novas relações
        for combinacao in combinacoes:
            pessoa1, pessoa2, relacao = combinacao

            # Verificar se podemos adicionar essa combinação à permutação atual
            if pessoa1 not in pessoas_na_permutacao or pessoa2 not in pessoas_na_permutacao:
                nova_permutacao = list(permutacao_atual)
                nova_permutacao.append(combinacao)

                # Atualizar o conjunto de pessoas conectadas
                novas_pessoas_na_permutacao = set(pessoas_na_permutacao)
                novas_pessoas_na_permutacao.add(pessoa1)
                novas_pessoas_na_permutacao.add(pessoa2)

                # Chamada recursiva para gerar mais combinações
                gerar(nova_permutacao, novas_pessoas_na_permutacao)

    # Iniciar com todas as pessoas e sem combinações
    gerar([], set())

    return permutacoes_possiveis
```

Análise Sintática do Código Iterado

```python
import colorama
from colorama import Fore, Back, Style

colorama.init()

tam_familia = int(input("Digite o tamanho da sua familia: "))

familia = {}
familia_ordem={}

vermelho = '\033[31m'
verde = '\033[32m'
azul = '\033[34m'
pink ='\033[1;35m'

reset = '\033[0;0m'

for i in range (tam_familia):
    nome = str(input(f"digite o nome do {i+1}º integrante: "))
    idade = int(input("digite  idade dele: "))
    familia[f'{nome}'] = idade

for i in sorted(familia, key = familia.get, reverse = True):
    familia_ordem[f'{i}'] = familia[i]

print(familia_ordem)

lista_familia = list(familia_ordem.items())

for i in range(len(lista_familia)):
    nome_now, idade_now = lista_familia[i]
    nomeM, idadeM = lista_familia[0]
```

```python
for j in range(len(lista_familia)):
    if i != j:
        nome_next, idade_next = lista_familia[j]

        diferenca = idade_now - idade_next

        if diferenca > 15 and diferenca < 50:
            print( vermelho + nome_now + reset + " pode ser pai/mãe de " + nome_next )

        if diferenca < -15 and diferenca > -50:
            print( vermelho + nome_now + reset + " pode ser filho(a) de " + nome_next)

        if diferenca <= 15 and diferenca >= -15:
            print(pink + nome_now + reset + " e " + nome_next + " podem ser irmãos(ãs) ou casal")

        if diferenca <= -80:
            print(azul + nome_now + reset + " pode ser bisneto(a) de " + nome_next )

        if diferenca >= 80:
            print( azul + nome_now + reset + " pode ser bisavô/vó de " + nome_next)

        if diferenca < 80 and diferenca >= 50:
            print(verde + nome_now + reset + " pode ser avô/avó de " + nome_next)

        if diferenca <=-50 and diferenca > -80:
            print(verde + nome_now + reset + " pode ser neto(a) de " + nome_next)

        colorama.deinit()
```

Complexidade do Tempo

O que é complexidade de tempo (Big O)?

A complexidade de tempo (Big O) nos diz como o tempo de execução de um algoritmo cresce à medida que a entrada aumenta. No seu caso, estamos falando de como o código responde conforme você adiciona mais pessoas (membros da família).

O(1) significa que o tempo de execução é constante, ou seja, o tempo não muda independentemente do número de membros.

O(n) significa que o tempo de execução cresce linearmente, ou seja, o tempo aumenta proporcionalmente ao número de membros.

$O(n^2)$ significa que o tempo de execução cresce quadraticamente, ou seja, se o número de membros dobra, o tempo de execução pode quadruplicar.

O(n!) significa que o tempo de execução cresce muito rapidamente com o aumento da entrada, tornando o código extremamente lento para grandes números.

Complexidade do Tempo

definir_relacoes_possiveis

Essa função compara a idade de duas pessoas e decide quais relações familiares elas podem ter (pai/mãe, filho, etc.). Ela sempre faz a mesma quantidade de verificações, independentemente do número de pessoas na lista.

Complexidade: O(1) (tempo constante).

Isso significa que o tempo que a função leva para ser executada não muda dependendo do número de membros da família. Toda vez que ela é chamada, leva o mesmo tempo para rodar, independentemente da quantidade de pessoas.

Complexidade do Tempo

`gerar_combinacoes`

O que é $O(n^2)$?

Quando falamos em $O(n^2)$, estamos nos referindo ao fato de que o número de operações (ou comparações) cresce quadraticamente com o número de elementos. Isso geralmente acontece quando estamos comparando todos os pares possíveis de elementos em um conjunto.

Fórmula para Comparação de Pares

Se você tiver n pessoas, o número de comparações de pares possíveis é dado pela fórmula de combinação de 2 elementos entre n:

$$C(n,2) = n(n-1)/2$$

2 pessoas | 3 pessoas | 10 pessoas

Complexidade do Tempo

gerar_permutacoes_validas

Por que $O(n^2!)$?

A complexidade de $O(n^2!)$ (fatorial quadrático) vem do fato de que estamos gerando todas as permutações de combinações de relações.

combinações de pares

A função gerar_combinacoes gera todas as possíveis combinações de pares entre os membros. Como explicado anteriormente, o número de combinações é $O(n^2)$, ou seja, para n pessoas, o número de pares possíveis cresce quadraticamente.

permutações de combinações

Agora, para cada conjunto de combinações gerado, a função gerar_permutacoes_validas tenta organizar essas combinações em todas as formas possíveis de conectar as pessoas. Isso envolve testar todas as permutações dessas combinações.

Complexidade do Tempo

arvore_genealogica

Complexidade $O(n^2!)$

A função arvore_genealogica é dominada pelo tempo de execução das funções que ela chama, especialmente a função gerar_permutacoes_validas, que, como vimos, tem complexidade $O(n^2!)$.

porquê?

Como o tempo de execução da função arvore_genealogica depende principalmente do tempo de execução de gerar_permutacoes_validas, a complexidade geral da função será a mesma: $O(n^2!)$.

Complexidade do Tempo

definir_relacoes_possiveis	O(1)
gerar_combinacoes	$O(n^2)$
gerar_permutacoes_validas	$O(n^2!)$
arvore_genealogica	$O(n^2!)$

No geral, a complexidade mais impactante é a da função gerar_permutacoes_validas, que é $O(n^2!)$ no pior caso, tornando este algoritmo pouco eficiente para um grande número de membros.

Complexidade do Tempo

```
tam_familia = int(input("Digite o tamanho da sua familia: "))
```

O que é O(1)?

significa que o tempo de execução é constante, ou seja, o tempo não muda independentemente do número de membros.

Este Input é uma informação necessário para todo o seguimento do programa pois é dele que tiraremos as repetições e membros da familia para serem permutados e calculados seus parentescos

O(1)

Complexidade do Tempo

```
# definir membros da familia

for i in range (tam_familia):
    nome = str(input(f"digite o nome do {i+1}º integrante: "))
    idade = int(input("digite  idade dele: "))
    familia[f'{nome}'] = idade
```

O que é O(n)?

significa que o tempo de execução cresce linearmente, ou seja, o tempo aumenta proporcionalmente ao número de membros.

Esse FOR exibira as perguntas e recebera respostas do nome e idade dos familiares, de acordo com o numero estipulado de familiares que o usuario esclareceu no inicio da execução.

O(n)

Complexidade do Tempo

```
# ordena os familiares j, em ordem decrescente
for i in sorted(familia, key = familia.get, reverse = True):
    familia_ordem[f'{i}'] = familia[i]

print(familia_ordem)

lista_familia = list(familia_ordem.items())
```

O que é O(n log(n))?

O(n log(n)) é uma notação que indica a complexidade de um trecho de código que possui um laço interno O(n) e um laço externo O(log(n)).

O FOR fará a função de organizar o dicionario de informações dos familiares de forma decrscente. A quantidade de vezes execurtadas serão multiplicadas de acordo com a quantidade de familiares inseridos.

→

O(n log(n))

Complexidade do Tempo

```python
for i in range(len(lista_familia)):
    nome_now, idade_now = lista_familia[i]
    nomeM, idadeM = lista_familia[0]

    for j in range(len(lista_familia)):
        if i != j:
            nome_next, idade_next = lista_familia[j]

            diferenca = idade_now - idade_next

            if diferenca > 15 and diferenca < 50:
                print( vermelho + nome_now + reset + " pode ser pai/mãe de " + nome_next )

            if diferenca < -15 and diferenca > -50:
                print( vermelho + nome_now + reset + " pode ser filho(a) de " + nome_next)

            if diferenca <= 15 and diferenca >= -15:
                print(pink + nome_now + reset + " e " + nome_next + " podem ser irmãos(ãs) ou casal")

            if diferenca <= -80:
                print(azul + nome_now + reset + " pode ser bisneto(a) de " + nome_next )

            if diferenca >= 80:
                print( azul + nome_now + reset + " pode ser bisavô/vó de " + nome_next)

            if diferenca < 80 and diferenca >= 50:
                print(verde + nome_now + reset + " pode ser avô/avó de " + nome_next)

            if diferenca <=-50 and diferenca > -80:
                print(verde + nome_now + reset + " pode ser neto(a) de " + nome_next)

            colorama.deinit()
```

Neste caso o codigo em que vemos executara a função de realizar as permutações dos familiares definidos que estao guardados no dicionario do codigo. Com dois FOR'S passamos por posições e comparamos com seus familiares em seguinte, determinando seu parentesco.

$O(n^2)$

Complexidade do Tempo

Quantidade de familiares	**O(1)**
Definir os membros	**O(n)**
Ordenar o dicionario de info	**O(n log(n))**
Permutações de parentesco	**O(n²)**

No geral, a complexidade mais impactante é a "Permutação de parentesco", na qual o O(n²) pode acarretar em um tempo mais elevado do que as outras ações, mas ainda sendo capaz de realizar uma grande quantidade de informações.

Recursividade X Iteração

Em vista de nosso caso de negócio, a recursividade não é tão adequada para ser utilizada, já que ela pode não performar muito bem e acabar sendo muito custosa e demorada a realização das permutações do parentesco da família. Assim não sendo a melhor opção.

Em vista do nosso caso de negócio, a iteração é bem mais adequada que a recursividade, já que a análise de parentesco e as permutações de uma família podem ser muito extensas e densas, por conta do volume das informações que são disponibilizadas. Assim prevalecendo a eficiência.

Obrigado pela atenção de todos!

codigos disponiveis em: AgataCeci, Enzo-dorta, sarahamelo, VIcente-VP