Desafio: Sequência de Fibonacci Recursiva

A sequência de Fibonacci é uma série de números onde cada número é a soma dos dois anteriores. A sequência começa com 0 e 1. Ou seja:

- F(0) = 0
- F(1) = 1
- F(n) = F(n-1) + F(n-2) para n > 1

Crie uma função recursiva que calcule o n-ésimo número de Fibonacci. Depois, escreva um programa que peça para o usuário inserir um número inteiro (n) e imprima o valor de F(n).

Exemplo de Entrada e Saída:

Entrada:

In [ ]:
```
Digite um número: 6
```

Saída:

In [ ]:
```
O número Fibonacci na posição 6 é: 8
```

In [17]:
```python
def fibonacci(number):
    if number == 0 or number == 1:
        return number
    else:
        return fibonacci(number - 1) +  fibonacci(number - 2)
```

In [20]:
```python
# Solicitar ao usuário um número
n = int(input("Digite um número: "))

# Calcular e exibir o resultado
print(f"O número Fibonacci na posição {n} é: {fibonacci(n)}")
```

```
O número Fibonacci na posição 6 é: 8
```

O código pode ser otimizado com memoization, podemos usar um dicionário para armazenar os resultados dos cálculos anteriores, evitando a recomputação dos mesmos valores várias vezes. Uma maneira simples de implementar isso em Python é usar o decorator `functools.lru_cache`, que faz cache automaticamente dos resultados das chamadas de função.

In [21]:
```python
from functools import lru_cache

@lru_cache(None)  # Usando cache ilimitado
def fibonacci(number):
    if number == 0 or number == 1:
```

```python
        return number
    else:
        return fibonacci(number - 1) + fibonacci(number - 2)

# Solicitar ao usuário um número
n = int(input("Digite um número: "))

# Calcular e exibir o resultado
print(f"O número Fibonacci na posição {n} é: {fibonacci(n)}")
```

```
O número Fibonacci na posição 78 é: 8944394323791464
```

Explicação:

1. **@lru_cache(None)**: O decorador `lru_cache` armazena os resultados das chamadas da função. O parâmetro `None` significa que o cache pode armazenar um número ilimitado de valores. Com isso, quando você calcular `fibonacci(5)` por exemplo, o resultado será armazenado e reutilizado se a função for chamada novamente para esse número.

2. **Eficiência**: Com a memoização, o algoritmo não precisa recalcular o valor de Fibonacci para números já computados, o que reduz a complexidade de tempo de O(2^n) para O(n).

Essa técnica é muito eficaz, especialmente quando você precisa calcular Fibonacci para valores grandes, como 1000 ou mais!

In [ ]: