Atividade 5 de desenvolvimento de código otimizado

Allan Garcia Cavalcante e Silva, 13731222
Eduarda Fritzen Neumann, 12556973
Lucas Eduardo Gulka Pulcinelli, 12547336
Teo Sobrino Alves, 12557192

Dezembro de 2023

1 Descrição da atividade

O código analisado foi o cálculo da função de Ackermann, notória por seu rápido crescimento. Ela foi proposta como um exemplo de uma função recursiva que não é prontamente expressa por meio de funções computáveis mais simples. Tal função, geralmente denotada como A(m, n), depende de dois parâmetros inteiros não negativos, m e n. Sua definição é dada por casos recursivos:

$$A(m,n) = \begin{cases} n+1 & \text{se } m = 0 \\ A(m-1,1) & \text{se } m > 0 \text{ e } n = 0 \\ A(m-1, A(m,n-1)) & \text{se } m > 0 \text{ e } n > 0 \end{cases}$$

Para isso, foram feitas duas implementações do cálculo de A(m, n), uma delas utilizando a otimização memoization e uma não. O decorador *lru_cache* da biblioteca *functools* foi utilizado para alcançar esse objetivo, configurada para guardar, no máximo, o resultado de 128 chamadas.

```
def ackermann(m, n):
    if m == 0:
        result = n + 1
    elif m > 0 and n == 0:
        result = ackermann(m - 1, 1)
    elif m > 0 and n > 0:
        result = ackermann(m - 1, ackermann(m, n - 1))

    return result

@lru_cache(maxsize=128)
def ackermann_memoization(m, n):
    if m == 0:
        result = n + 1
```

```
    elif m > 0 and n == 0:
        result = ackermann_memoization(m - 1, 1)
    elif m > 0 and n > 0:
        result = ackermann_memoization(m - 1, ackermann_memoization(m, n - 1))

    return result
```

Para a medida do tempo de execução foi utilizada a biblioteca *timeit*, por meio do método *repeat* da classe *Timer*. Para a criação dos objetos *Timer* utilizando as funções implementadas, que tomam parâmetros, foi utilizado a função *partial* de *functools*, fixando os parâmetros m=3 e n=5 para todos os testes. Os testes foram realizados 10 vezes, cada vez executando a função 100 vezes.

```
timer_memoization = timeit.Timer(partial(ackermann_memoization, 3, 5))
timer_normal = timeit.Timer(partial(ackermann, 3, 5))

results_memoization = timer_memoization.repeat(repeat=10, number=100)
results_normal = timer_normal.repeat(repeat=10, number=100)
```

2 Resultados e análise

Tabela 1: Resultado dos experimentos

	Tempo de execução (s)	
Experimento	Com otimização	Sem otimização
0	0.000501979000092	1.086817194999639
1	0.000014054999610	1.664141661000031
2	0.000015349000023	1.614517623999745
3	0.000013080999906	1.034925416000078
4	0.000015961999907	0.820396187999904
5	0.000016560000404	0.804914673999974
6	0.000016880000203	0.799753000999772
7	0.000017116999970	0.807436395999957
8	0.000016712000161	0.800538641000003
9	0.000012822999906	0.806515474000207

A média dos tempos de execução do código sem otimizações foi 1.023996 s, com desvio padrão de 0.323486 s (cerca de 32% da média), e do código com otimização foi 0.000064 s, com desvio padrão de 0.00015 s (cerca de 228% da média). Portanto, a otimização memoization reduziu o tempo médio de execução do código aproximadamente 15987 vezes. O desvio padrão referente ao código com *memoization* foi consideravelmente maior, percentualmente, do que o referente ao código sem essa otimização, o que pode ser devido ao tempo

de execução ser ordens de grandeza menor, logo qualquer alteração no ambiente de execução tem um impacto maior no tempo de execução.

3 Conclusão

Pode-se perceber que a técnica de otimização *memoization* reduziu consideravelmente o tempo de execução do código. Isso era esperado, já que o cálculo da função de Ackermann A(m, n) depende do próprio valor da função para valores menores de m e n, cenário ideal para a otimização utilizada.