Relatório - Atividade 5

Artur Kenzo - 15652663
Daniel Jorge - 15446861
Gabriel Phelippe - 15453730
Jhonatan Barboza - 15645049

November 3, 2025

1 Introdução

Este relatório apresenta uma análise de desempenho do uso de uma técnica de otimização específica para a aplicação em códigos Python. Optamos por aplicar a técnica de memoização (do inglês, memoization) através do decorador `"@functools.lru_cache"`, uma ferramenta da biblioteca padrão do Python que armazena em cache os resultados de chamadas de função.

2 Metodologia

2.1 Arquitetura

- Processador: 12th Gen Intel(R) Core(TM) i5-1235U x 12
- Linux GNU/Debian
- Interpretador: Python 3.11.2 (compilado com GCC 12.2.0)

2.2 Programas selecionados

Para a aplicação da ferramenta e coleta dos dados, utilizamos códigos que buscam resolver um problema de programação dinâmica usando unicamente "força bruta", com recursão e apresentando uma complexidade de $\mathbf{O(2^n)}$—um com a ferramenta de otimização aplicada e o outro, sem. Numa análise extra, também temos um terceiro código que resolve o problema com uma abordagem dinâmica, que foi comparado com os outros dois. Os códigos podem ser encontrados aqui.

2.3 Ferramentas utilizadas

2.3.1 Técnica de Otimização

A técnica de otimização empregada nesta atividade foi a memoização (do inglês, *memoization*), implementada através do decorador `@functools.lru_cache` da

biblioteca padrão do Python. Este decorador é uma forma de "metaprogramação" que envolve a função alvo — neste caso, a função recursiva `resolver_forca_bruta` — e adiciona a ela a funcionalidade de cache.

2.3.2 Medição de Tempo

Para comparar os tempos de execução dos códigos, foi utilizada a biblioteca `timeit`. Dela, fez-se uso da função `timeit.default_timer()`, que fornece, durante a execução de código, o tempo em que foi chamada. Assim, ela é chamada anterior e posteriormente à função de resolução do problema e, ao fim, os tempos registrados são subtraídos, obtendo o tempo de execução.

3 Resultados

Os experimentos foram feitos executando os três códigos (Força Bruta, Otimizado com `@lru_cache` e Programação Dinâmica Manual) em três casos de teste com tamanhos de entrada (N) crescentes. Cada execução foi repetida três vezes.

Os tempos médios de execução para cada abordagem e caso de teste estão registrados na Tabela 1.

Table 1: Comparação dos tempos médios de execução (em segundos) por abordagem.

Caso de Teste (N)	Força Bruta (sem otimização)	Otimizado (`@lru_cache`)	PD Manual (extra)
N = 8	0.000250 s	0.000117 s	0.000058 s
N = 14	0.010424 s	0.000237 s	0.000126 s
N = 22	1.779923 s	0.000711 s	0.000282 s

Nota: Os tempos são a média de 3 execuções. Os tempos individuais estão disponíveis nos arquivos de resultado no mesmo repositório dos códigos.

3.1 Análise dos Resultados

A partir dos dados coletados na Tabela 1, várias observações podem ser feitas:

- **Validade dos Resultados:** Todos os algoritmos, independentemente da otimização, produziram as saídas esperadas corretas para os casos de teste (17, 43 e 64), validando a lógica da implementação.
- **Custo da Força Bruta:** A versão de força bruta (sem otimização) sofre uma degradação de performance exponencial, como previsto pela sua complexidade $\mathbf{O(2^n)}$. O tempo de execução salta de 0.000250s (N=8) para 0.010424s (N=14), e então escala para 1.779923s (N=22).
- **Eficácia da Memoização:** A aplicação do decorador `@functools.lru_cache` se mostrou extremamente eficaz. No caso de teste N=22, a otimização reduziu o tempo de 1.779923s para apenas 0.000711s, uma melhoria de mais de 2500 vezes. Isso demonstra que a função recursiva de fato possuía muitos subproblemas repetidos, que agora são calculados apenas uma vez.

- **Comparação com PD Manual:** Uma observação interessante é a comparação entre a memoização automática do `@lru_cache` e a implementação de Programação Dinâmica manual (que usa uma tabela de memoização explícita). Em todos os casos, a versão manual foi mais rápida. No caso N=22, a versão de PD Manual foi aproximadamente 2.5x mais rápida que o `@lru_cache` (0.000282s contra 0.000711s). Isso sugere que, embora o `@lru_cache` seja uma ferramenta de otimização poderosa e fácil de aplicar, ele carrega um custo de *overhead* que uma solução manual e especializada não possui.

4 Conclusões

Com os resultados obtidos, é possível obter uma visão geral sobre a comparação das performances dos códigos e das técnicas associadas a eles:

4.1 "forca_bruta.py"

Ficou evidente que a solução de força bruta, embora funcional para entradas muito pequenas (N=8), se torna inviável rapidamente. O salto para quase 2 segundos de execução em um caso de teste ainda não muito grande (N=22) confirma a complexidade $\mathbf{O(2^n)}$ e a necessidade de otimização.

4.2 "forca_bruta_otimizado.py"

Nesse sentido, o decorador `@functools.lru_cache` obteve ótimos resultados. Com a adição de uma única linha de código (e a conversão de listas para tuplas), o desempenho do algoritmo melhorou consideravelmente, reduzindo o tempo de execução de 1.78s para 0.0007s no caso mais crítico.

4.3 "dinamico.py" (extra)

Por fim, é interessante apontar como o `@lru_cache`, apesar de conveniente, demonstrou ter um certo *overhead* (custo computacional) em comparação com uma solução manual de programação dinâmica, presente neste código, que se mostrou ainda mais rápida.

Concluímos, então, que as ferramentas de otimização da biblioteca padrão do Python, como o `@lru_cache`, são extremamente poderosas e práticas, permitindo que o desenvolvedor resolva grandes gargalos de performance sem a necessidade de reescrever manualmente a lógica do algoritmo.