Programa de Pós-graduação - Mestrado em Modelagem Computacional
Disciplina: Algoritmos I / GA-026
Quarto período de 2023

Trabalho Individual - Relatório Final.
Tema: Programação Dinâmica - Problema de subsequência mais longa & Multiplicação cadeias de matrizes.

Discente: Arthur Henrique Craveiro Costa
Docente: Roberto Pinto Souto

Petrópolis, RJ.
18 de dezembro de 2023.

Sumário

Sumário.....**1**
Ambiente Computacional.....**2**
O Problema de Subsequência mais Longa.....**3**
1. Introdução do algoritmo.....3
2. Previsão teórica do big O e comprovação experimental.....4
3. Exemplos práticos.....6
Multiplicação de Cadeias de Matrizes.....**7**
1. Introdução do algoritmo.....7
2. Previsão teórica do big O e comprovação experimental.....9
3. Exemplos práticos.....11
Referências bibliográficas.....**12**
Anexos.....**13**

Ambiente Computacional

Antes de iniciarmos os dois tópicos que compõem o tema do trabalho, segue abaixo a descrição o mais detalhada possível do ambiente computacional onde os experimentos de ambos os tópicos foram tratados.

Eles foram realizados utilizando o Google Colab. Consultando as informações do ambiente temos os seguintes dados retornados:

- Sistema Operacional: Windows 10.
- Disco: 26.2/107.7 GB.
- Memória RAM: 1.4/12.7 GB.
- Processador (CPU): Intel(R) Xeon(R) CPU @ 2.20GHz.
- Linguagem de Programação: Python 3.

Observação: Foram utilizados os comandos !df -h; !cat /proc/cpuinfo e !cat /proc/meminfo em células do notebook para obter-se essas informações.

O Problema de Subsequência mais Longa

1. Introdução do algoritmo.

O Problema de subsequência mais longa crescente (Longest Increasing Subsequence - LIS) é um problema da ciência da computação que dado uma sequência de números $(a_1, a_2, \ldots, a_n)$ consiste em encontrar uma subsequência na qual os números estão ordenados em ordem crescente e esta é a maior possível (máxima). Um exemplo rápido que pode ser visto no livro “Algoritmos” de 2009 do Dasgupta, página 157, é a sequência 5, 2, 8, 6, 3, 6, 9, 7 , onde sua LIS é 2, 3, 6, 9:

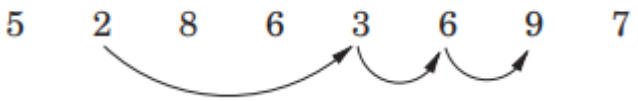


Figura 1: Máxima subsequência crescente.

Esse problema pode ser visto como um DAG (Directed Acyclic Graph) onde cada vértice é um número da sequência dada e as arestas são transições possíveis de um vértice para o outro, onde uma aresta (i, j) que liga dois números só é possível se o primeiro for menor que o segundo, isto é $a_i < a_j$ e $i < j$. Para o exemplo acima, o livro também ilustra o DAG das subsequências crescentes.

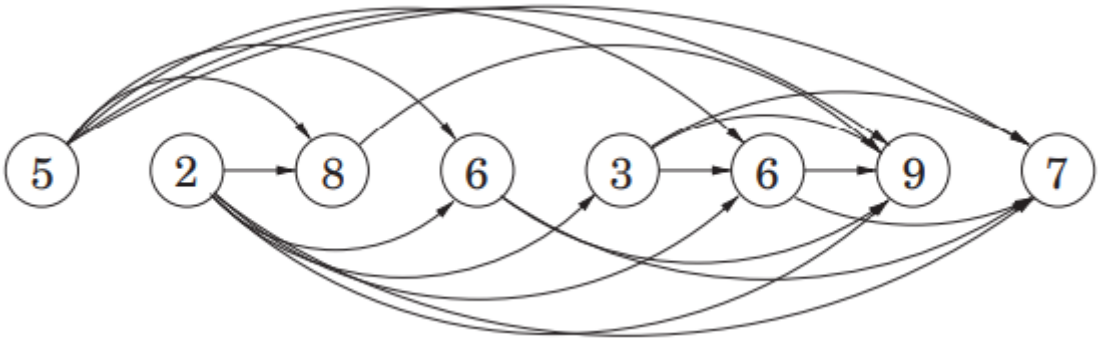


Figura 2: O DAG das subseqüências crescentes.

Então o problema consiste, basicamente, em encontrar o caminho mais longo no DAG. Para resolver esse problema, foi utilizado o código em anexo, extraído e adaptado do site GeekforGeeks (Longest Increasing Subsequence (LIS) - GeeksforGeeks).

Vamos entender melhor a função “def longest_increasing_subsequence(arr)”:

1. Inicialização:
   a. Cria uma lista vazia chamada “lis” para armazenar subsequências crescentes até cada índice na sequência de entrada.
2. Inicialização da Primeira Subsequência:
   a. Inicializa a primeira subsequência com o primeiro elemento da sequência de entrada.
3. Percorrer os Elementos:
   a. Usa dois loops para percorrer os elementos da sequência.
   b. Para cada elemento, compara-o com os elementos anteriores.

c. Se o elemento atual for maior que o elemento anterior e se estender a subsequência até esse ponto for benéfico, atualiza a subsequência atual.

4. Encontrar a Subsequência mais Longa:
   a. Após preencher a tabela lis, encontra a subsequência mais longa usando a função max com base no comprimento.
5. Retorno:
   a. Retorna a subsequência mais longa.

2. Previsão teórica do big O e comprovação experimental.

Dada uma sequência de entrada com n números, a previsão teórica do limite superior de tempo (ou complexidade algorítmica) é de $O(n^2)$, que pode ser visto claramente pelos dois loops for aninhados que percorrem os elementos da sequência:

```
for i in range(1, n):
    for j in range(0, i):
        if arr[i] > arr[j] and len(lis[i]) < len(lis[j]) + 1:
            lis[i] = lis[j] + [arr[i]]
```

Figura 3: Loops FOR na função LIS.

Enquanto o loop externo (na variável i) percorre cada elemento da sequência uma única vez (complexidade $O(n)$), o loop interno (na variável j) percorre os elementos anteriores a i, no pior caso percorrendo $O(n)$ elementos (quando i é $n-1$). Logo a complexidade total do loop interno é $O(n^2)$ em todo o algoritmo.

Isso significa que o tempo de execução do algoritmo cresce quadraticamente em relação ao tamanho da entrada. Em termos de eficiência, este algoritmo pode ser impraticável para sequências muito grandes.

Falando mais especificamente sobre cada um dos casos.

O pior caso ocorre quando se tem uma lista ordenada em ordem crescente como sequência de entrada, o que seria o pior caso para o algoritmo em questão, pois ele sempre teria que percorrer toda a lista para encontrar a subsequência mais longa (verificar a condição para todos os pares possíveis de índices i e j), resultando em uma complexidade de $O(n^2)$.

Já o melhor caso ocorre quando é inserida uma lista ordenada em ordem decrescente (ou constante), o que seria o melhor caso para este algoritmo, porque o loop interno verificaria apenas uma condição para cada i. Isso ainda resulta em uma complexidade de $O(n^2)$, porém é possível diminuir para $O(n)$, fazendo uma verificação antecipada no início do algoritmo:

```
if all(arr[i] >= arr[i + 1] for i in range(n - 1)):
    return [arr[0]]
```

Figura 4: Alteração na execução do melhor caso.

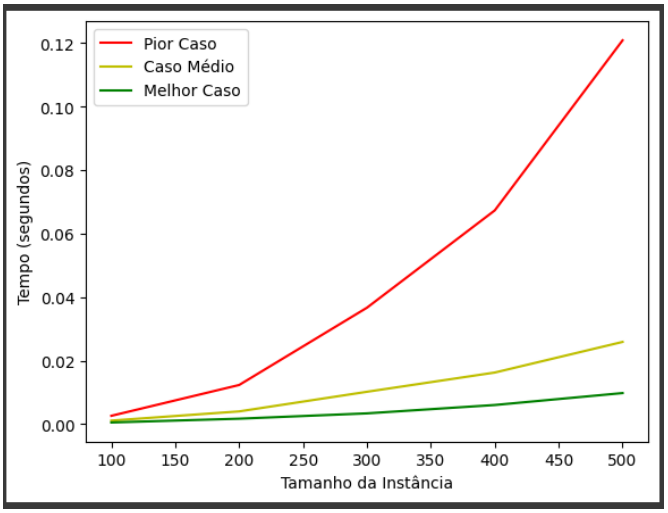
Esta verificação rápida determina se a lista está em ordem decrescente. Se for o caso, o algoritmo retorna imediatamente a maior subsequência crescente (que é

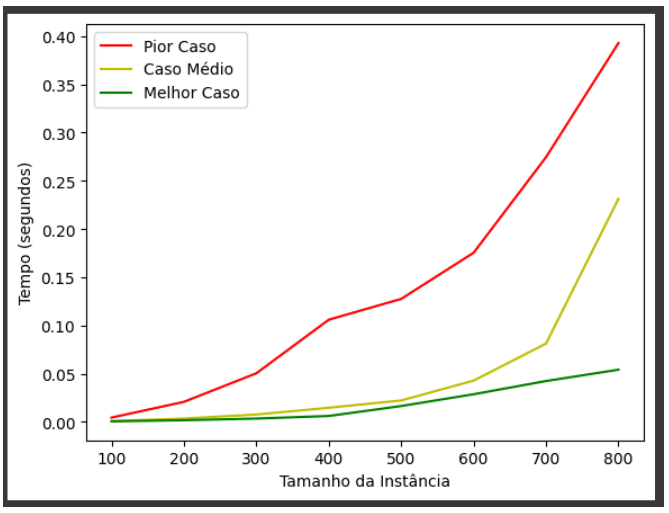
simplesmente o primeiro elemento da lista). Assim, em vez de percorrer todos os elementos da lista duas vezes (complexidade $O(n^2)$), o algoritmo agora percorre a lista apenas uma vez para realizar essa verificação, resultando em uma complexidade de tempo $O(n)$ para o melhor caso.

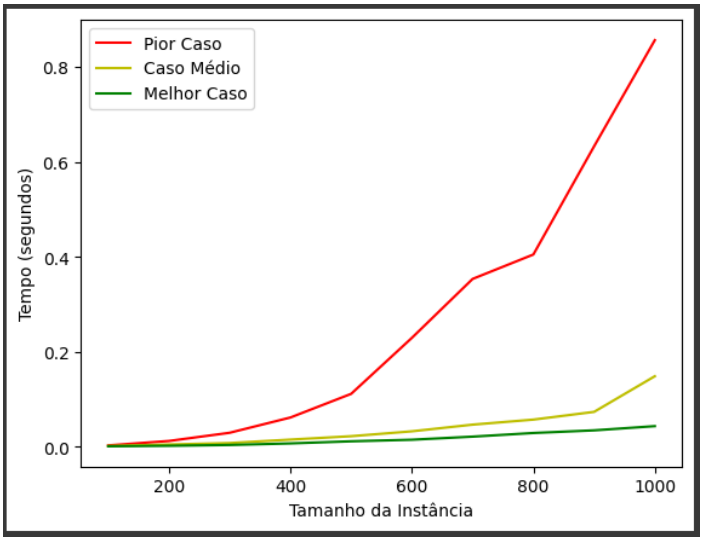
Sobre o caso médio, no geral, é considerado $O(n^2)$, assim como no pior caso, porque o algoritmo ainda precisa realizar um número significativo de comparações em muitos pares possíveis de índices i e j, mesmo quando a entrada não está estritamente ordenada em ordem crescente. No código, é gerada uma lista com números entre 1 e 10000 vezes o tamanho do teste atual, de forma aleatória, de acordo com o tamanho de cada teste. É importante destacar que a forma de gerar essa lista pode alterar bastante o tempo neste caso, porém ele tende a permanecer entre os dois acima quase sempre.

Dessa forma, tendo em mente, que são geradas listas de "tamanhos" diferentes, cada uma criada de acordo com a ideia de gerar cada um desses casos da melhor maneira possível, ficamos com o seguinte gráfico como resultado dos testes (em geral, pois em cada teste podem haver mudanças)

Nos gráficos a seguir é possível ter uma ideia de que a ordem de crescimento do pior caso e do caso médio é quadrática enquanto o melhor caso segue uma linha linear, olhando o código sem a alteração no melhor caso.







Figuras 5, 6 e 7: exemplos, da esquerda para a direita, de cima para baixo, com instância máxima de 500, 800 e 1000 e com valores no caso médio variando até 1000, 2000 e 10000 respectivamente.

Com a alteração na complexidade do melhor caso, feita através da verificação anterior aos loops aninhados, a curva dele diminuiria, como podemos ver no exemplo ao lado:

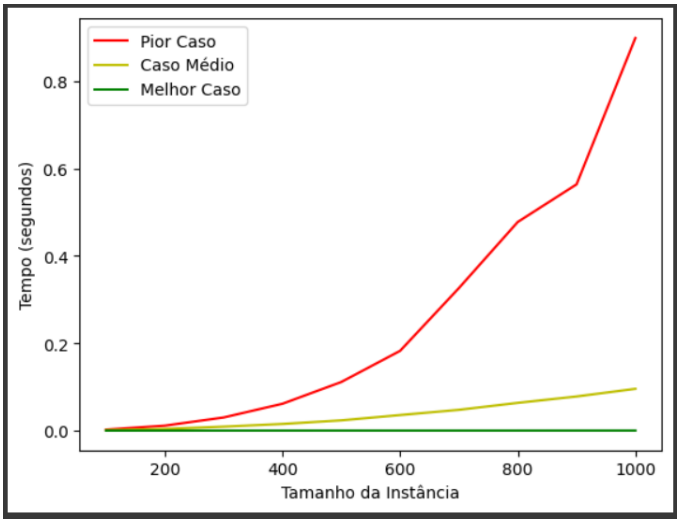


Figura 8: Gráficos de exemplo com alteração no melhor caso.

3. Exemplos práticos.

A importância do Problema da Subsequência Mais Longa Crescente (LIS) se estende além da teoria, encontrando aplicação em diversos campos práticos. Um exemplo notável é a análise de dados em bioinformática, onde o LIS é utilizado no alinhamento de sequências de DNA para identificar regiões conservadas entre diferentes espécies. Este processo é fundamental para entender a evolução e as funções genéticas.

Outra aplicação prática do LIS está na otimização de sistemas de recomendação. Por exemplo, em plataformas de streaming, o LIS pode ajudar a identificar padrões de visualização dos usuários para recomendar conteúdo que se alinha com suas preferências sequenciais. Isso melhora a experiência do usuário e aumenta a eficiência do sistema de recomendação.

Além disso, o LIS encontra aplicações em áreas como análise financeira e processamento de linguagem natural, demonstrando sua versatilidade e relevância em uma variedade de contextos práticos. Ele não é diretamente aplicado em análise financeira ou processamento de linguagem natural (NLP), mas sim o conceito de encontrar padrões sequenciais em dados, que é central nesse algoritmo, pode inspirar abordagens em NLP e análise financeira, especialmente em tarefas como análise de tendências ou identificação de padrões em séries temporais. Nessas áreas, algoritmos semelhantes ao LIS podem ser utilizados para discernir sequências significativas em grandes conjuntos de dados, embora o algoritmo aqui propriamente dito não seja diretamente aplicado.

Multiplicação de Cadeias de Matrizes

1. Introdução do algoritmo.

O problema da multiplicação de cadeias de matrizes é um problema de otimização que envolve encontrar a melhor ordem para multiplicar uma sequência de matrizes de forma a minimizar o número total de operações escalares. Este problema é um exemplo clássico de programação dinâmica, onde soluções de subproblemas menores são combinadas para resolver o problema maior. O algoritmo se baseia em particionar a cadeia de matrizes e, através de uma abordagem de "dividir para conquistar", calcular o custo mínimo de multiplicação para cada possível divisão da cadeia. Vale ressaltar que o problema envolve apenas decidir a maneira mais eficiente de realizar a sequência de multiplicações, e não elas em si.

Como a multiplicação de matrizes é associativa (não comutativa), existem inúmeras formas de colocar parênteses entre as matrizes e o resultado será o mesmo. Por exemplo, dadas 4 matrizes, $A, B, C\ e\ D$, de diferentes dimensões, como pode ser visto no livro “Algoritmos” de 2008 do Dasgupta, página 168, e na figura abaixo, existem diferentes formas de organizar os parênteses para efetuar as multiplicações.

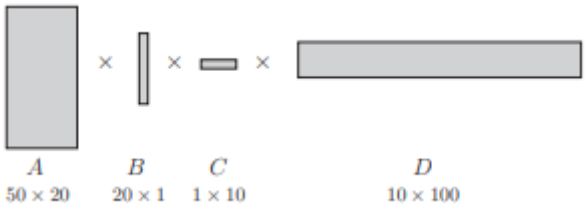


Figura 9 :Dimensões de A, B, C e D.

Multiplicar uma matriz $m\,x\,n$ por uma outra matriz $n\,x\,p$ toma mnp mutiplicações, então dependendo da forma que os parênteses são posicionados, um número diferente de operações é realizada:

Organização de parênteses	Computação do custo	Custo
$A \times ((B \times C) \times D)$	$20 \cdot 1 \cdot 10 + 20 \cdot 10 \cdot 100 + 50 \cdot 20 \cdot 100$	120.200
$(A \times (B \times C)) \times D$	$20 \cdot 1 \cdot 10 + 50 \cdot 20 \cdot 10 + 50 \cdot 10 \cdot 100$	60.200
$(A \times B) \times (C \times D)$	$50 \cdot 20 \cdot 1 + 1 \cdot 10 \cdot 100 + 50 \cdot 1 \cdot 100$	7.000

Figura 10: Algumas formas de organizar os parênteses na multiplicação de 4 matrizes.

Então, se queremos multiplicar $A_1 \times A_2 \times \ldots \times A_n$ matrizes com dimensões $m_0 \times m_1$, $m_1 \times m_2$, … , $m_{n-1} \times m_n$, respectivamente, podemos construir uma árvore binária onde cada matriz é uma folha, a raiz é o produto final e os nós interiores são os produtos intermediários. Cada árvore corresponde a uma possível organização de parênteses e o objetivo é achar uma árvore ótima para resolver os problemas. Ela, por sua vez, possui subárvores ótimas.

(a) $((A \times B) \times C) \times D$; (b) $A \times ((B \times C) \times D)$; (c) $(A \times (B \times C)) \times D$.

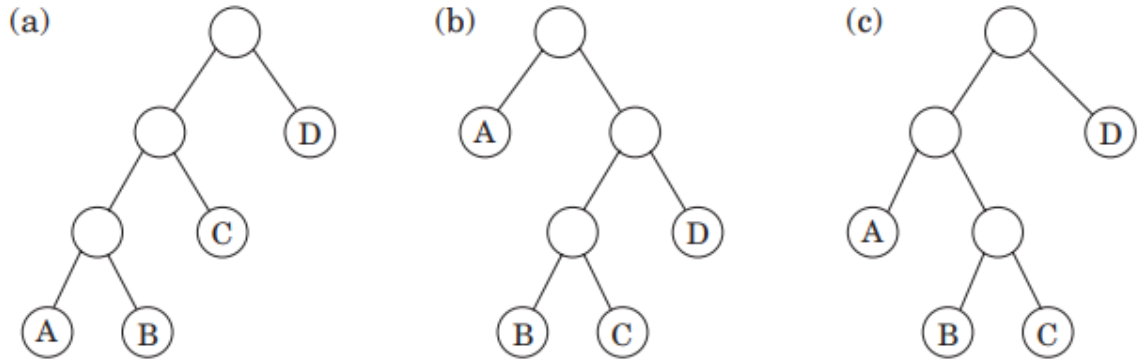


Figura 11: Árvores binárias possíveis para solução do problema.

Assim, para resolver esse problema, foi utilizado o código em anexo, extraído e adaptado do site GeekforGeeks (Matrix Chain Multiplication | DP-8 - GeeksforGeeks).

Vamos entender melhor a função “def OrdemMultiplicacaoMatriz(arr, tamanho)”:

1. Inicialização:
   a. A função começa inicializando uma matriz vazia m de tamanho $n\,x\,n$ (onde n é o tamanho da sequência de matrizes) para armazenar os custos mínimos de multiplicação entre as matrizes.
2. Inicialização dos Custos Iniciais:
   a. Define os valores na diagonal principal da matriz m para 0, pois o custo de multiplicar uma única matriz é sempre zero.
3. Percorrer os Tamanhos de Subcadeias:
   a. Utiliza dois loops aninhados para percorrer diferentes tamanhos de subcadeias, começando com subcadeias de comprimento 2 e aumentando gradualmente até subcadeias de comprimento n.
   b. O loop externo, representado por L, determina o tamanho da subcadeia que estamos considerando.
4. Percorrer as Subcadeias:
   a. O loop interno, representado por i, itera sobre todas as posições possíveis de início da subcadeia de tamanho L.
   b. Com base em i, calcula a posição final da subcadeia como $j = i + L - 1$.
   c. Inicializa o custo mínimo da subcadeia atual como infinito ($maximo\_int$).
5. Calculando o Custo Mínimo para a Subcadeia:
   a. Dentro do loop interno, utiliza um terceiro loop, representado por k, para iterar entre i e $j - 1$.
   b. Calcula o custo da multiplicação das matrizes da subcadeia atual, considerando todas as possíveis posições de divisão entre i e j, e compara esse custo com o custo mínimo atual.
6. Atualização do Custo Mínimo:

a. Se o custo calculado for menor do que o custo mínimo atual, atualiza o custo mínimo para esse valor calculado.

7. Preenchimento da Tabela:
   a. Após o término dos loops L e i, a matriz m estará preenchida com os custos mínimos de multiplicação para todas as sub-cadeias possíveis.
8. Encontrando o Custo Mínimo Total:
   a. O valor em $m[1][n-1]$ contém o custo mínimo total de multiplicação para todas as matrizes na sequência.
9. Retorno:
   a. A função retorna o custo mínimo total encontrado, que representa o número mínimo de multiplicações necessárias para multiplicar todas as matrizes na ordem ótima.

2. Previsão teórica do big O e comprovação experimental.

Dada uma sequência de entrada com n matrizes encadeadas, previsão teórica do limite superior de tempo (ou complexidade algorítmica) é de $O(n^3)$, que pode ser visto claramente pelos 3 loops aninhados abaixo:

```
for L in range(2, n):
    for i in range(1, n - L + 1):
        j = i + L - 1
        m[i][j] = maximo_int
        for k in range(i, j):
            custo = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j]
            if custo < m[i][j]:
                m[i][j] = custo
```

Figura 12: Loops FOR na função OrdemMultiplicacaoMatriz.

Para cada par de matrizes na cadeia (definido pelos índices i e j), o algoritmo calcula o custo mínimo de multiplicação, o que envolve uma iteração sobre todos os possíveis pontos de divisão da cadeia (índice k). Esta iteração é feita para cada par de matrizes na cadeia.

Olhando para os 3 loops aninhados, podemos ver que:

- O loop externo que varia o comprimento da cadeia de matrizes (L);
- O loop intermediário que escolhe o início da cadeia (i);
- O loop interno que escolhe o ponto de divisão da cadeia (k).

Como cada loop pode iterar até n vezes, o número total de operações é proporcional a $n \times n \times n = O(n^3)$. O código calcula de forma eficiente o número mínimo de multiplicações escalares necessárias, armazenando e reutilizando resultados intermediários, que é uma característica fundamental da programação dinâmica para otimizar o desempenho.

Falando especificamente sobre os casos, diferentemente de muitos outros algoritmos, não apresenta uma variação significativa de complexidade entre o melhor caso, o pior caso e o caso médio. Em todos esses casos, a complexidade de tempo permanece $O(n^3)$. A razão para essa uniformidade na complexidade de tempo é que o algoritmo explora

todas as possíveis maneiras de agrupar as matrizes para a multiplicação, independentemente das dimensões específicas das matrizes. Ele calcula e armazena o custo mínimo de multiplicação para todos os segmentos possíveis da cadeia de matrizes, garantindo que a solução ótima seja encontrada. Este processo é realizado de forma completa para todos os possíveis segmentos da cadeia, levando à mesma complexidade de tempo em todos os casos.

Isso pode ser visto nos gráficos abaixo, onde cada caso foi escolhido da seguinte forma:

- Melhor Caso: Dimensões uniformes, o que poderia teoricamente simplificar as multiplicações.
- Pior Caso: Alternamos entre dimensões grandes e pequenas, o que pode aumentar a complexidade das multiplicações.
- Caso Médio: Dimensões aleatórias.

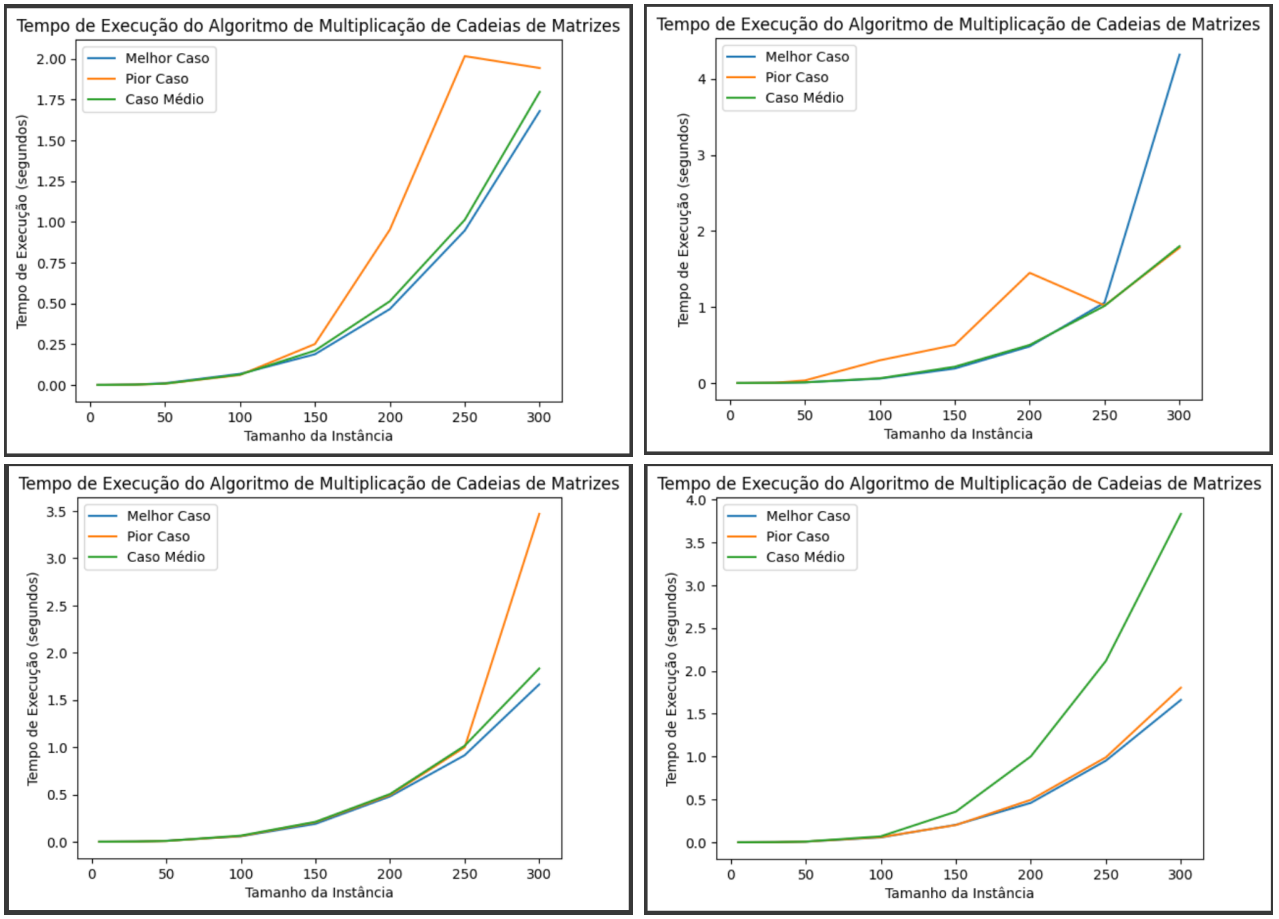


Figura 13: Alguns testes experimentais com tamanho de instância máximo de 300.

Como podemos ver na imagem acima, alguns testes seguem a ideia teórica dos casos e outros não, devido a dificuldade de diferenciar os casos, já que como o algoritmo de explora todas as possíveis divisões da cadeia de matrizes, as variações nas dimensões das matrizes podem não afetar significativamente o tempo de execução como esperado.

3. Exemplos práticos.

Um exemplo prático da aplicação do algoritmo de Multiplicação de Cadeias de Matrizes está na otimização de cálculos em redes neurais. Em redes neurais profundas, diversas matrizes de pesos são multiplicadas durante o processo de feedforward e backpropagation. A ordem eficiente de multiplicação dessas matrizes pode reduzir significativamente o tempo de cálculo.

Outra aplicação está na computação gráfica, especialmente em animações e jogos, onde múltiplas transformações matriciais são aplicadas para modelar, rotacionar e escalar objetos 3D. A multiplicação eficiente de cadeias de matrizes pode otimizar essas transformações, melhorando o desempenho e a eficiência dos cálculos gráficos.

Estes exemplos demonstram a relevância do algoritmo em campos que requerem processamento intensivo de matrizes, destacando sua importância na otimização de operações complexas em computação moderna.

Referências bibliográficas.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). "Introduction to Algorithms" (3ª ed.). MIT Press.

Dasgupta, S., Papadimitriou, C., & Vazirani, U. (2009). "Algoritmos" (tradução técnica por Prof. Dr. Guilherme Albuquerque Pinto). McGraw-Hill.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). "Deep Learning". MIT Press.

Hughes, J. F., Van Dam, A., McGuire, M., Sklar, D. F., Foley, J. D., Feiner, S. K., & Akeley, K. (2013). "Computer Graphics: Principles and Practice" (3ª ed.). Addison-Wesley.

"GeeksforGeeks." [Online]. Disponível em: https://www.geeksforgeeks.org/.

Anexos.

Links para os códigos de utilizados ao longo do trabalho e apresentados em sala de aula:

1. Problema de Subsequência mais longa ;
2. Multiplicação de Cadeias de Matrizes .