Trabalho Prático 0  
Operações com matrizes alocadas dinamicamente

**Carlos Leão Henrique Brito Malta Leão**

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG) Belo Horizonte – MG – Brasil

# Introdução

[chbmleao@](mailto:chbmleao@)ufmg.br

Esta documentação lida com o problema proposto de implementação de um Tipo Abstrato de Dados (TAD) de matrizes, em que suas duas dimensões são alocadas dinamicamente. Além disso, esse TAD deve realizar três operações sobre matrizes: soma, multiplicação e transposição, desenvolvendo a abstração, desempenho e robustez do programa. Esta documentação tem como objetivo explicitar como foi realizada a implementação desse TAD, além de realizar uma análise explicativa de cada uma das três operações sobre matrizes de forma individual.

Para resolver o problema citado, foi criado um programa na linguagem C, que utiliza uma estrutura chamada *matriz* que implementar o TAD e operações solicitadas, que aloca os elementos da matriz de forma dinâmica para evitar gastos de memória desnecessários. Por fim, ao decorrer dessa documentação, alguns aspectos sobre o trabalho serão melhor explicados, como a implementação do código na seção 2 e as instruções de compilação e execução na seção 3. Já na seção 4, será feita uma análise de complexidade mais detalhada sobre as três operações principais do programa.

# Implementação

# Organização

Em relação à organização do código, foi dividido seguindo as instruções do trabalho prático, em que a pasta TP é a raiz do projeto, a pasta ‘bin’ está vazia, src armazena arquivos de código na linguagem c (‘\*.c’), a pasta ‘include’ contém os cabeçalhos do projeto (‘\*.h’) e, por fim, a pasta ‘obj’ está vazia. Além disso, o Makefile gera os códigos objeto (‘\*.o’) no diretório ‘obj’ e o executável do TP no diretório ‘bin’.

Ademais, para melhor organização do projeto, existem duas pastas, ‘assets’, que contém algumas matrizes de entrada em arquivos de texto (‘\*.txt’), e a pasta ‘out’, que inicialmente fica vazia, mas com a execução do Makefile, recebe as matrizes de saída, também em arquivos de texto.

O projeto apresenta três arquivos principais, primeiramente o cabeçalho (/include/mat.h), que contém a implementação da estrutura *‘mat\_tipo’*, que contém os atributos e funções do TAD matriz. Em segundo lugar, temos o arquivo (/src/mat.c), que implementa todas as funções do *‘mat\_tipo’.* Por fim, temos o arquivo (/src/matop.c), que é o código Main do programa, responsável por interpretar os comandos do Makefile e do usuário e executar uma das três operações de matrizes, soma, multiplicação ou transposição.

# Estruturas de dados e funcionamento do programa

A principal estrutura de dados utilizada foi o Tipo Abstrato de Dados, *matriz\_tipo,* implementado utilizando *structs*, registros em C, quem implementa os atributos de uma matriz, como as suas dimensões x e y, número de identificação e seus elementos.

Nesse registro também são implementadas todas as funções, sendo as principais *somaMatrizes*, *multiplicaMatrizes* e *transpoeMatriz*. Além dessas três funções, também são implementadas algumas funções auxiliares, como *criaMatrizTxt* (lê um arquivo txt que apresenta as informações de uma matriz), *inicializaMatrizNula* (inicializa todos os elementos de uma matriz como zero), *acessaMatriz* (acessa a matriz para fins de registro de acesso), *escreveArquivoTxt* (escreve a matriz em um arquivo txt) e, por fim, *destroiMatriz* (destroi a matriz, que se torna inacessível).

O programa principal, de forma sucinta, baseia-se na função main do arquivo ‘src/matop.c’, que utiliza a função *parse\_args* para receber os valores passados pelo usuário para a execução do programa e atribuição de variáveis. Dessa forma, uma das três seguintes opções terá sido escolhida pelo usuário:

* **OPSOMA:** Lê dois arquivos txt que contém as matrizes A e B. Assim, as matrizes a e b são somadas e formam a matriz C, que é escrita também em um arquivo txt. Por fim, todas as matrizes são destruídas.
* **OPMULTIPLICA:** Lê dois arquivos txt para a formação das matrizes A e B. Assim, as matrizes A e B são multiplicadas e formam a matriz C, que é escrita também em um arquivo txt. Por fim, todas as matrizes são destruídas.
* **OPTRANSPOR:** Lê um arquivo txt para a formação da matriz A. Dessa forma, a matriz A é transposta, escrita em um arquivo txt, e, por fim, destruída.

# Especificações técnicas

* Sistema operacional: WSL Ubuntu 20.04
* Linguagem de programação implementada: C
* Compilador: gcc (Ubuntu 9.3.0-17ubuntu1)
* Processador: Intel Core i5-9400F CPU @ 2.90GHz 2.90 GHz
* Memória RAM: 02 X Win Memory 8GB, DDR4, 2666MHZ

# Instruc¸ões de compilação e execução

# Primeiramente, utilizando um terminal, acesse o diretório TP.

# Ao digitar o comando *make all*, o arquivo Makefile será executado no terminal, e, dessa forma, realizará todas as seguintes operações, que também podem ser executadas de maneira separada (*make perf*, por exemplo):

# *mem:* Executa a soma, multiplicação e transposição utilizando matrizes quadradas com dimensões 5 por 5, utilizando as matrizes ‘m1\_5.txt’ e ‘m2\_5.txt’ localizadas na pasta *assets*. Por fim, as matrizes de saída são armazenadas na pasta *out*. Além disso, os acessos a memória são registrados e utilizados para a plotagem de gráficos utilizando a biblioteca disponibilizada *analisamem.* Os gráficos são disponibilizados na pasta *tmp*, em que os gráficos da soma, multiplicação e transposição são disponibilizados nas pastas *sumadin*, *multdin*, *transdin*, respectivamente.

# *perf: Executa também* a soma, multiplicação e transposição utilizando matrizes quadradas com dimensões que variam de 1000 por 1000 para 500 por 500, variando de 100 em 100. Da mesma forma que a execução *mem*, os arquivos são retirados da pasta *assets*, e armazenados os resultados na pasta *out*. Além disso, os resultados de desempenho são armazenados por essa função, que retorna os tempos de execução de cada operação sobre cada matriz. Os resultados de desempenho também são armazenados na pasta *tmp*, para ver os resultados, basta acessar a pasta tmp no terminal e escrever comandos no seguinte formato: ‘*more mult1000.out*’.

# *gprof:* Executa também as três operações de matrizes, utilizando matrizes quadradas com dimensões de 1000 por 1000. Ademais, utiliza o gprof para criar o perfil raso e grafo de execução de cada operação, que são armazenadas em arquivos na pasta *tmp* dessa forma: *‘tmp/soma1000gprof.txt’*.

# *clean:* Limpa o diretório de objetos *obj*, excluindo os arquivos *mat.o, matop.o* e *memlog.o*. Além disso, exclui também o arquivo *gmon.out*, localizado na raíz do programa.

# *all:* Executa os comandos *mem*, *perf* e *gprof*.

# Caso o usuário perfira, é possível executar apenas uma operação utilizando matrizes de sua escolha. Para isso, é preciso executar uma das seguintes linhas de comando, substituindo os nomes dos arquivos txt para os arquivos desejados:

# Soma: ‘bin/matop -s 1 m1.txt -2 m2.txt -o res.txt’

# Multiplicação: ‘bin/matop -m 1 m1.txt -2 m2.txt -o res.txt’

# Soma: ‘bin/matop -s 1 m1.txt -o res.txt’

# Em caso de dúvidas, é possível executar o comando ‘bin/matop -h’ para visualizar a mensagem de uso do programa.

# Análise de complexidade

Esta seção apresenta a análise de complexidade de tempo e espaço para as três principais operações sobre matrizes do programa, soma, multiplicação e transposição. Como foi supracitado na seção 2. Para um melhor entendimento, faremos a análise de cada um dos três procedimentos de forma individual.

Para realizar as análises, utilizaremos a variável para representar as dimensões da matriz, ou seja, é igual a ou , mesmo que essas variáveis não sejam necessariamente iguais. A análise será feita dessa forma para simplificar e facilitar o entendimento geral. Além disso, muitas funções são utilizadas em mais de um procedimento, dessa forma, para a leitura não se tornar repetitiva, cada função será analisada apenas uma vez. Por fim, a análise de complexidade de espaço será realizada de duas formas, primeiramente analisando as funções utilizadas uma a uma, e depois o processo todo em conjunto, analisando os gráficos gerados pela execução do Makefile, o que garantirá um melhor entendimento sobre a utilização da memória.

# SOMA

# Durante o procedimento de soma, existem 3 fases:

# Durante a primeira fase, as seguintes funções são chamadas: *criaMatrizTxt* (2 vezes, para inicializar a matriz A e B), *criaMatriz* (1 vez para inicializar a matriz C) e *inicializaMatrizNula* (1 vez para inicializar C como uma matriz nula).

# Durante a segunda fase, as seguintes funções são chamadas: *acessaMatriz* (3 vezes para fins de registro das três matrizes) e *somaMatrizes* (1 vez, realiza a soma da matriz A e B).

# Por fim, na terceira fase, as seguintes funções são chamadas: *acessaMatriz*(1 vez, para acessar a matriz C), *escreveArquivoTxt* (1 vez, escreve a matriz C) e, por último, *destroiMatriz* (3 vezes, destrói as três matrizes).

# Complexidade de tempo:

# *criaMatriz:* Inicializa todos os atributos da struct *mat\_tipo*, . Além disso aloca todos os elementos da matriz de dimensão x passando por dois laços aninhados, . Dessa forma, teremos a complexidade assintótica da função como:

# *criaMatrizTxt:* Essa função chama a função *criaMatriz,* , além disso, percorre todo o arquivo txt da matriz e atribui cada elemento para a matriz da struct, passando também por dois laços aninhados . Dessa forma, teremos a complexidade assintótica da função como:

# *inicializaMatrizNula*: Inicializa todos os atributos da struct *mat\_tipo* como nulos, passando por dois laços aninhados Dessa forma, teremos a complexidade assintótica da função como:

***acessaMatriz***: Acessa todos os elementos da matriz x , passando por dois laços aninhados, . Dessa forma, teremos a complexidade assintótica da função como:

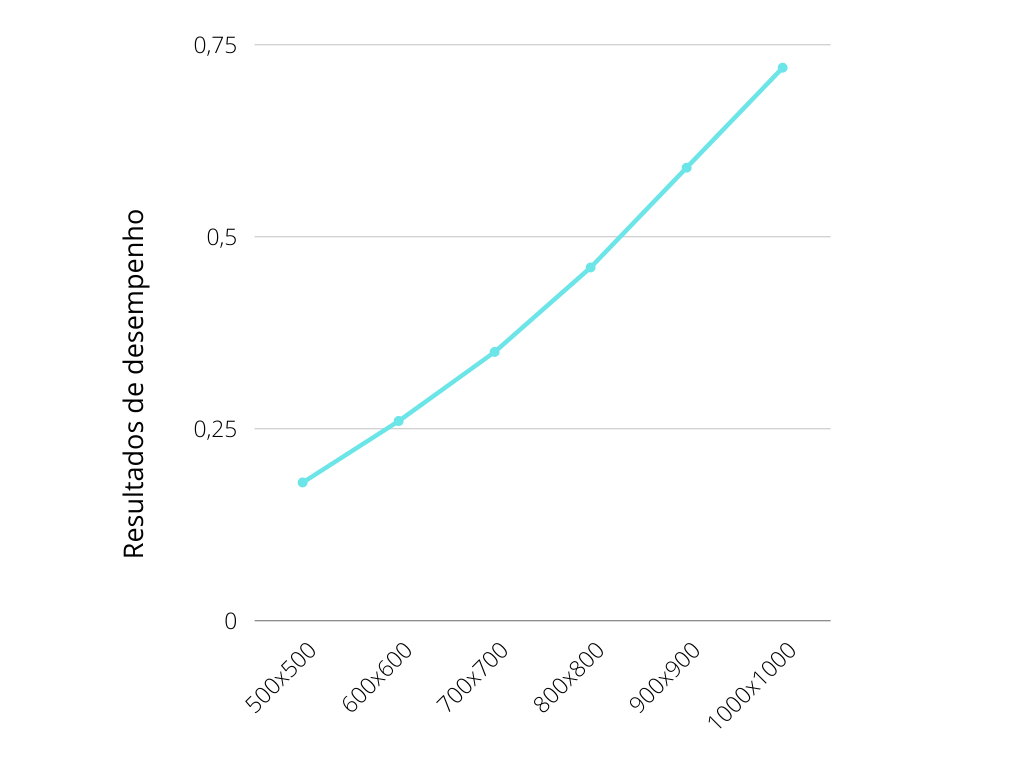
***somaMatrizes***: Chama as funções *criaMatriz*e*inicializaMatrizNula,* . Atribui todos os elementos da matriz C, passando por dois laços aninhados, . Dessa forma, teremos a complexidade assintótica da função como:

# *escreveArquivoTxt*: Lê todos os elementos da matriz C, passando por dois laços, . Dessa forma, teremos a complexidade assintótica da função como:

# *destroiMatriz:* Passa por apenas um laço, por todas as linhas da matriz, . Dessa forma, teremos a complexidade assintótica da função como:

# Por fim, podemos realizar a análise completa do procedimento de soma, somando todas a complexidades assintóticas de cada função multiplicado pelas vezes que são usadas.

Assim, concluímos que a complexidade assintótica do procedimento de soma de matrizes é . Isso pode ser comprovado ao observar o gráfico de resultados de desempenho de cada matriz utilizada, de dimensões de 500x500 até 1000x1000, em que o gráfico é visivelmente exponencial.



# Complexidade de espaço:

Para realizar a análise de complexidade de espaço, primeiramente faremos uma análise de cada função utilizada no procedimento de soma, assim como realizamos na análise de complexidade de tempo. Em seguida, observaremos os gráficos gerados pela execução do Makefile para um melhor entendimento do uso da memória durante toda a execução. Nesse sentido, todos os gráficos utilizam matrizes aleatórias de dimensão 5x5. Além disso, são gerados gráficos de acesso e evolução da distância de pilha para cada uma das matrizes utilizadas na função e os gráficos de distância de pilha, além de serem divididos para cada matriz, também são divididos em três fases (fases que já foram supracitados na análise de complexidade de tempo, nessa mesma seção 4).

# *criaMatriz:* Utiliza variáveis auxiliares para alocar os atributos da matriz, , e aloca uma matriz com linhas e colunas, generalizando, teremos elementos . Dessa forma, teremos a complexidade de espaço da função como:

# *criaMatrizTxt:* Essa função chama a função *criaMatriz,* , além disso, percorre todo o arquivo de texto da matriz e atribui cada elemento para a matriz já alocada, . Ademais, algumas variáveis também são utilizadas para realizar algumas operações . Dessa forma, teremos a complexidade de espaço da função como:

# *inicializaMatrizNula*: Inicializa todos os atributos da struct *mat\_tipo* como nulos, passando por elementos, . Dessa forma, teremos a complexidade de espaço da função como:

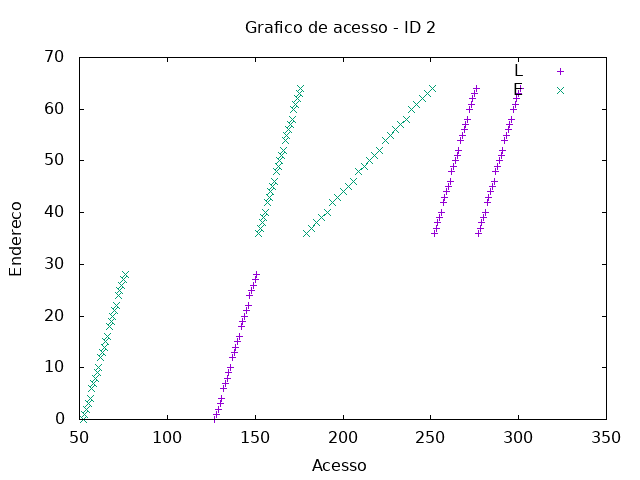
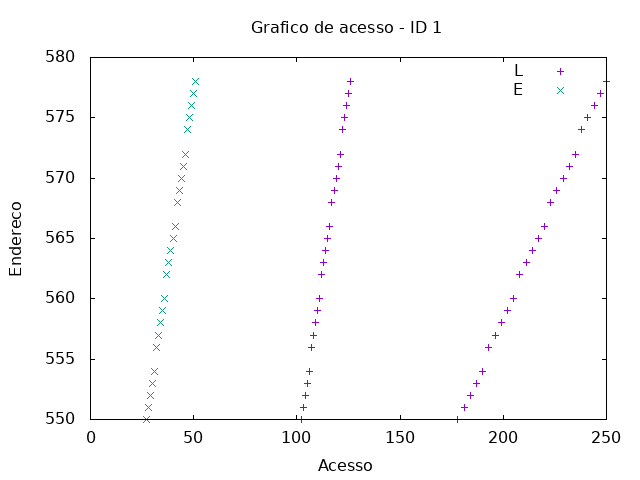
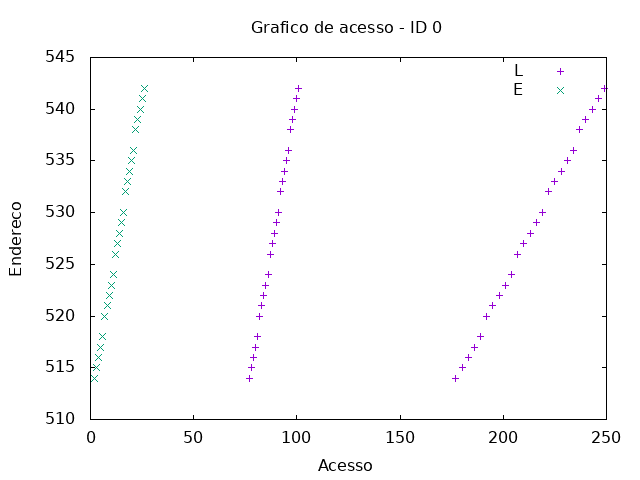
***acessaMatriz***: Acessa todos os elementos da matriz x , passando por dois elementos, . Dessa forma, teremos a complexidade assintótica da função como:

***somaMatrizes***: Chama as funções *criaMatriz*e*inicializaMatrizNula,* . Atribui todos os elementos da matriz C, ou seja, elementos, . Dessa forma, teremos a complexidade assintótica da função como:

# *escreveArquivoTxt*: Realiza todas as operações considerando estruturas auxiliares unitárias e a matriz de entrada de tamanho , que é passada como parâmetro da função . Assim, a complexidade de espaço da função é:

# *destroiMatriz:* Realiza todas as operações considerando estruturas auxiliares unitárias . Além disso, percorre todos os endereços de memória das linhas da matriz, para limpar a memória de cada uma separadamente, .

# Por fim, podemos realizar a análise completa da complexidade de espaço do procedimento de soma, somando todas a complexidades encontradas de cada função multiplicado pelas vezes que são usadas.



# Estamos analisando uma soma de matrizes no formato , dessa forma, cada um dos três gráficos acima representa uma matriz, ou seja, A = ID0, B = ID1, C = ID2. Os gráficos informam o endereço acessado em determinado acesso, em que os acessos são realizados um após o outro. Além disso, existe uma diferenciação entre um simples acesso, que é representado na cor roxa, e uma atribuição de valor, que é representado na cor verde. Dito isso, podemos realizar a análise dos gráficos.

# Primeiramente, temos no primeiro e segundo gráfico as matrizes A e B, que apresentam um comportamento muito semelhante, como já foi visto na análise de tempo. Primeiro a matriz é preenchida com os valores de um arquivo txt, e em seguida é acessada pela função *acessaMatriz*, o que é representado no gráfico pelas primeiras linhas de escrita e leitura. Depois disso, existe mais uma linha de acesso, em que são acessados os valores das duas matrizes para realizar a operação de soma.

# Em segundo lugar, temos a matriz C, que possui um gráfico distinto das matrizes A e B. Primeiramente, a função *inicializaMatrizNula* é chamada e inicializa todos os elementos de C, e logo em seguida a função acessaMatriz também é chamada, gerando as linhas verde e roxa. Em seguida, a função de soma das matrizes é chamada, por questão de segurança a matriz C é inicializada com zeros, e, posteriormente cada elemento da matriz C é atribuído o valor da soma das matrizes A e B, formando a linha verde um pouco mais horizontal que a outra (a linha é um pouco mais horizontal devido ao acesso de um elemento da matriz A, um elemento da matriz B e em seguida a escrita do elemento da matriz C, um processo que se repete até o final da função). Por último, a matriz é acessada novamente duas vezes, uma pela função *acessaMatriz* e outra pela função *escreveArquivoTxt*.

# Z:\tmp\somadin\somadin-hist-0-0.png

# Z:\tmp\somadin\somadin-hist-0-1.png

# Z:\tmp\somadin\somadin-hist-0-2.png

# Z:\tmp\somadin\somadin-hist-1-0.png

# Z:\tmp\somadin\somadin-hist-1-1.png

# Z:\tmp\somadin\somadin-hist-1-2.png

# Z:\tmp\somadin\somadin-hist-2-2.png

# Agora analisaremos os gráficos de distância de pilha, em que além de serem divididos entre as três matrizes, também são divididos entre as três fases do procedimento de soma.

# Primeiramente, observando os três gráficos da primeira fase, percebemos que todas as distâncias de pilha das três matrizes são zero. Isso ocorre já que na primeira fase essas matrizes estão sendo inicializadas pelas funções *criaMatrizTxt* e *inicializaMatrizNula*, começando todas pelo primeiro endereço da matriz e atribuindo valores até o final. Como um endereço está sempre ao lado do outro, a distância de pilha é zero.

# Durante a segunda fase, os gráficos tornam-se mais distintos. Primeiramente, as matrizes A e B também apresentam um comportamento muito semelhante, já que são acessados pelas mesmas funções. Nesse sentido, primeiramente todos os elementos são acessados pela função acessaMatriz, em seguida, para acessar o primeiro elemento e realizar a soma, é preciso percorrer todas as 25 posições da matriz, o que gera uma distância de pilha igual a 25. Esse processo ocorre durante toda a função *somaMatrizes*. O gráfico da matriz C é um pouco diferente porque a função *inicializaMatrizNula* também é chamada, o que gera mais 25 distâncias de pilha iguais a zero.

# Por último, durante a terceira fase, apenas a matriz C é acessada, pela função *acessaMatriz* e *escreveArquivoTxt.* Esse acesso antecipado gera o mesmo efeito da segunda fase, que é preciso percorrer 25 elementos para escrever o arquivo de teto, o que gera uma distância de pilha igual a 25.

# somadin-distp-0

# Z:\tmp\somadin\somadin-distp-1.png

# Z:\tmp\somadin\somadin-distp-2.png

# Por fim, analisando os gráficos de evolução de distância de pilha, podemos reforçar os aspectos encontrados na análise dos gráficos de distância de pilha. Além disso, ao observar o gráfico da matriz C, notamos que primeiramente, durante a realização da soma, os acessos são um pouco mais espaçados, já que é necessário acessar um elemento da matriz A, um da matriz B, e depois atribuir o elemento na matriz C. Esse espaçamento não ocorre, por exemplo, durante a execução da função *escreveArquivoTxt*.

# MULTIPLICAÇÃO

# Durante o procedimento de multiplicação, também existem 3 fases:

# A primeira fase é muito semelhante a primeira fase da soma, em que se utiliza a função *criaMatrizTxt*, *criaMatriz* e *incializaMatrizNula*.

# Já a segunda fase, também chama a função *acessaMatriz*, para as três matrizes e *multiplicaMatrizes*, que realiza a multiplicação de A x B.

# Por último, a terceira fase também é muito semelhante ao procedimento de soma, em que são chamadas as funções de *acessaMatriz*, *escreveArquivoTxt* e *destroiMatriz*.

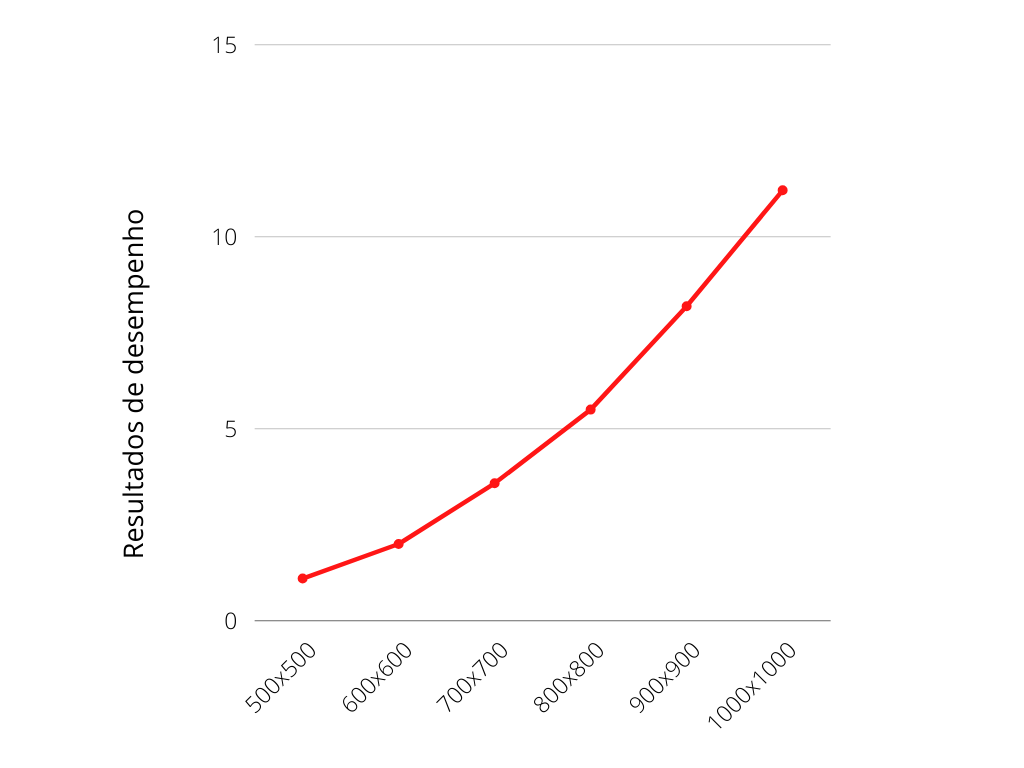
# Complexidade de tempo:

# O procedimento de multiplicação é muito semelhante ao procedimento de soma de matrizes. Dessa forma, quase todas as funções do procedimento de multiplicação de matrizes já tiveram sua complexidade assintótica determinada, apenas uma função ainda não foi analisada:

***multiplicaMatrizes***: Primeiramente, esta função realiza a chamada de outras, *criaMatriz*e*inicializaMatrizNula,* . Em seguida, para realizar a multiplicação das matrizes são necessários três laços aninhados (em A x B, é preciso associar cada linha de A com as colunas de B e realizar a multiplicação, cada linha de A será multiplicada por toda a matriz, elementos, e todas as linhas de A devem realizar o mesmo processo, ), . Dessa forma, teremos a complexidade assintótica da função como:

Por fim, podemos determinar a complexidade assintótica do procedimento de multiplicação de matrizes como um todo, assim como fizemos com o procedimento de soma. Dessa forma, multiplicamos cada complexidade assintótica de cada função pelo número de vezes que é chamada no processo e, por fim, somamos todas as complexidades:

# Portanto, concluímos que a complexidade assintótica do procedimento de multiplicação de matrizes é , maior que o procedimento de soma. Assim como fizemos no caso anterior, isso também pode ser notado ao analisar os resultados de desempenho dessa função, que apresenta valores temporais maiores e um crescimento mais acelerado que a função de soma.



# Complexidade de espaço:

# O procedimento de multiplicação é muito semelhante ao procedimento de soma de matrizes. Dessa forma, quase todas as funções do procedimento de multiplicação de matrizes já tiveram sua complexidade de espaço determinada, apenas uma função ainda não foi analisada:

***multiplicaMatrizes***: Essa função realiza todas as operações considerando estruturas auxiliares unitárias . Ademais, três matrizes são passadas como parâmetros e utilizadas, todas tendo dimensões de aproximadamente , . Dessa forma, teremos a complexidade de espaço da função como:

Por fim, podemos determinar a complexidade espacial do procedimento de multiplicação de matrizes como um todo, assim como fizemos com o procedimento de soma. Dessa forma, multiplicamos cada complexidade de cada função pelo número de vezes que é chamada no processo e, por fim, somamos todas as complexidades:

# 

# O procedimento da operação de multiplicação de matrizes também passa por três fases já supracitadas e utiliza três matrizes de ID 0 até ID 2, ou seja, . Todas as matrizes apresensentam dimensão de 5x5 e A e B são matrizes de números aleatórios. Dito isso, podemos realizar a análise de complexidade de espaço utilizando os gráficos gerados pela execução do arquivo Makefile e as bibliotecas *gprof* e *analisamem*.

# Z:\tmp\multdin\multdin-acesso-0.png

# Z:\tmp\multdin\multdin-acesso-1.png

# Z:\tmp\multdin\multdin-acesso-2.png

# Ao observar os três gráficos de acesso, percebemos que todos apresentam um aspecto em comum, uma função de escrita e outra de leitura. Nesse sentido, todas as matrizes são inicializadas e depois são acessados todos os valores atribuídos. Porém, é perceptível que após essa primeira parte, os acessos de cada uma das três matrizes são muito distintos.

# Partiremos agora para o uso da função *multiplicaMatrizes*, em que as linhas da matriz A são acessadas 5 vezes cada uma, enquanto as colunas da matriz B também são acessadas da mesma forma, isso ocorre porque o algoritmo de multiplicação de matrizes precisa multiplicar os elementos das linhas de A pelos elementos das colunas de B. Como cada linha precisa passar por todas as colunas de B, cada linha precisa ser percorrida 5 vezes.

# Por outro lado, a matriz C recebe todos os seus atributos novos, como em uma função de inicializar a matriz nula. Porém, a grande diferença que pode ser notada no gráfico de acesso, é que a alocação é realizada de uma maneira mais lenta e os elementos são escritos diversas vezes, isso ocorre já que é preciso percorrer toda uma linha de A e toda uma coluna de B para armazenar apenas um elemento de C, somando todos os valores, um de cada vez, o que gera 5 acessos seguidos à um mesmo endereço de memória. Por último todos os elementos de C são acessados duas vezes, pela função *acessaMatriz* e *escreveArquivoTxt*.

# Z:\tmp\multdin\multdin-distp-0.png

# Z:\tmp\multdin\multdin-distp-1.png

# Z:\tmp\multdin\multdin-distp-2.png

# Z:\tmp\multdin\multdin-hist-0-0.png

# Z:\tmp\multdin\multdin-hist-0-1.png

# Z:\tmp\multdin\multdin-hist-0-2.png

# Z:\tmp\multdin\multdin-hist-1-0.png

# Z:\tmp\multdin\multdin-hist-1-1.png

# Z:\tmp\multdin\multdin-hist-1-2.png

# Z:\tmp\multdin\multdin-hist-2-2.png

# Ao observar os gráficos de distância de pilha e de evolução da distância de pilha, podemos realizar algumas análises e alcançar algumas conclusões interessantes. Lembrando que existem três matrizes de diferentes números de identificação e três fases do processo de multiplicação. Dito isso, podemos realizar uma análise focada na distância de pilha do processo de multiplicação.

# Primeiramente, a fase 0 é muito semelhante ao procedimento de soma, em que a distância de pilha das três matrizes é zero, já que como ainda não foram inicializadas, o primeiro acesso começa com número de pilha zero. Dessa forma, todas as matrizes são preenchidas com seus devidos valores sem grandes gastos de memória.

# Durante a segunda fase, cada matriz apresenta um comportamento muito distinto das outras duas, começaremos analisando a matriz A. Ao observar o gráfico de distância de pilha da matriz A fica nítido que a maior parte dos acessos apresentam distância de pilha 5. Além disso, o gráfico de evolução nos mostra que primeiramente o acesso apresenta distância de pilha 25 e depois 5. Dessa forma, fica claro que no começo do acesso da linha a distância é 25, mas depois é preciso acessar a mesma linha novamente, o que retorna a distância de pilha igual a 5. Já o gráfico de distância de pilha da matriz B gera um pouco de dúvida, mas é melhor esclarecido ao analisar o gráfico de evolução. Nesse sentido, as colunas de B são acessadas uma a uma, porém, como o método *acessaMatriz* acessa os endereços de B linha a linha, os elementos de B estão “fora de ordem”, então ocorrem algumas inconsistências em relação a distância de pilha da matriz B, o que gera alguns números que em primeira mão não fazem muito sentido. Porém, após percorrer por completo todas as colunas de B, ao realizar o processo novamente, todas as distâncias de pilha serão iguais a 25.

# Por último, podemos analisar a matriz C de modo mais individual. Ainda durante a segunda fase, percebemos que prevalecem as distâncias de pilha iguais a 0, 25, e 1. As distâncias de pilha de 0 e 25 já foram explicadas em outros exemplos, a distância 0 ocorre quando o elemento seguinte da pilha é acessado e 25 ocorre quando é preciso percorrer 25 espaços na pilha até chegar ao elemento desejado. Porém, o número de pilha 1 não é muito comum, mas nesse caso ele é muito recorrente. Nesse sentido, o número 1 ocorre quando o mesmo endereço é acessado em sequência, o que ocorre no nosso código, já que cada elemento da matriz C é somado cinco vezes até chegar ao seu valor final. Por fim, durante a última fase as funções *acessaMatriz* e *escreveArquivoTxt* são chamadas, o que gera as distâncias de pilha 0 e 25, assim como analisado no procedimento de soma.

# 

# 

# TRANSPOSIÇÃO

# O procedimento de transposição é o mais distinto das três principais operações do programa, já que utiliza apenas uma matriz para realizar todos os processos. Porém, esse processo também apresenta três fases.

# A primeira fase apenas chama a função *criaMatrizTxt*, criando a matriz A.

# Já a segunda fase, também chama a função *acessaMatriz* e a função *transpoeMatriz*, passando a matriz A como parâmetro.

# Por último, a terceira fase é a que mais se assemelha aos procedimentos supracitados, em que é chamada a função *acessaMatriz*, *escreveArquivoTxt* e *destroiMatriz*, todas passando a matriz A como parâmetro.

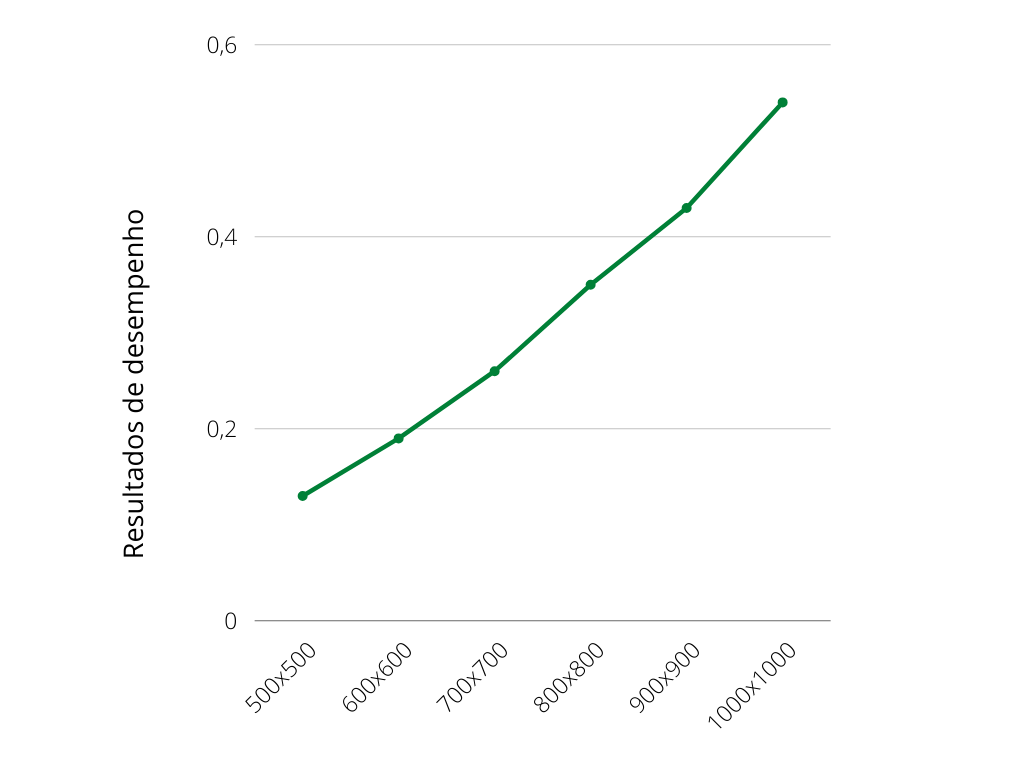
# Complexidade de tempo:

# A grande maioria das funções já tiveram sua complexidade assintótica encontrada. Dessa forma, basta encontrarmos a complexidade da última função.

# *transpoeMatrizes*: Primeiramente, esta função realiza a chamada da função criaMatriz, que cria uma matriz auxiliar, . Em seguida, para realizar a transposição da matriz são necessários dois laços aninhados (para copiar todos os elementos da matriz A e passar de forma transposta para a matriz auxiliar), . Por último a matriz A recebe o endereço dos atributos da matriz auxiliar, a deixando transposta, . Dessa forma, teremos a complexidade assintótica da função como:

Por fim, podemos determinar a complexidade assintótica do procedimento de transposição de matriz de forma geral. Dessa forma, multiplicamos cada complexidade assintótica de cada função pelo número de vezes que é chamada no processo e, por fim, somamos todas as complexidades:

Portanto, concluímos que a complexidade assintótica do procedimento de transposição de matriz é , o mesmo que o procedimento de soma e menor que o procedimento de multiplicação. Assim como fizemos nos outros casos, isso também pode ser notado ao analisar os resultados de desempenho dessa função, que apresenta valores menores que as outras duas (devido ao uso de apenas uma matriz e menor número de funções chamadas), mas um crescimento muito semelhante ao crescimento do processo de soma, que apresenta a mesma complexidade de tempo, .

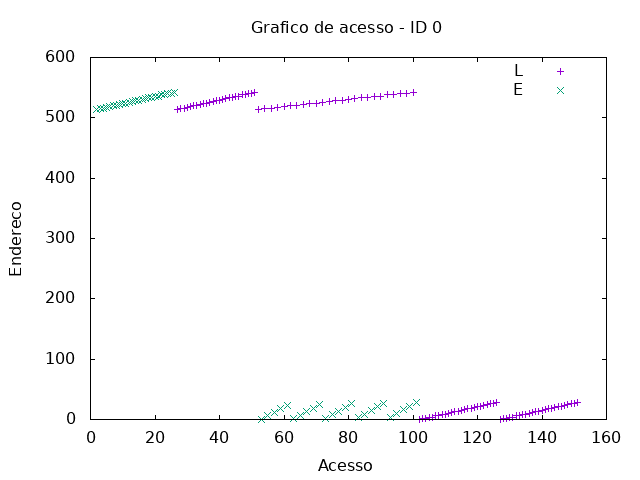


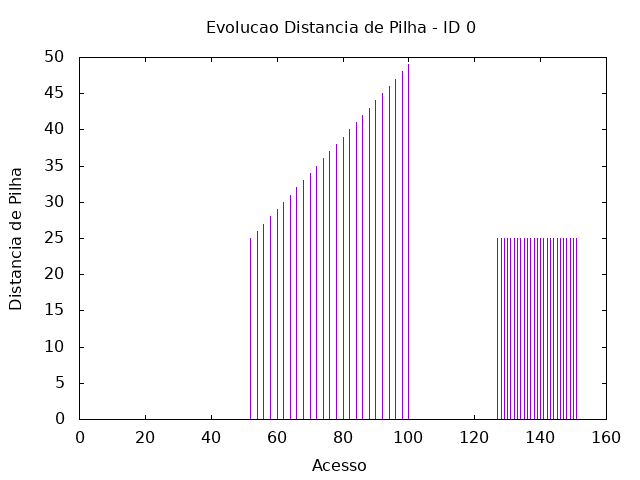
# Complexidade de espaço:

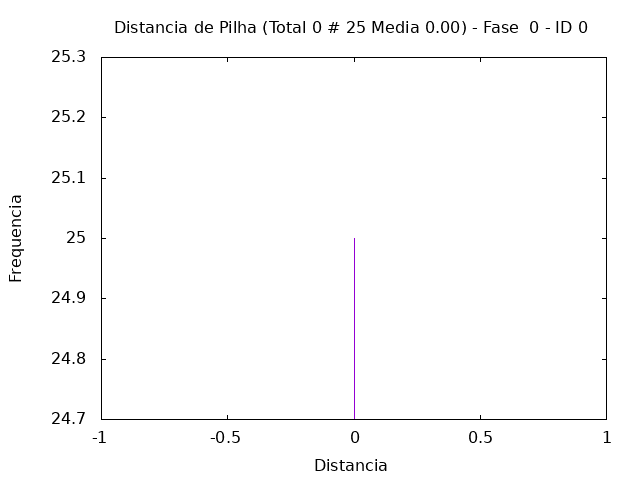
# A grande maioria das funções já tiveram sua complexidade espacial encontrada. Dessa forma, basta encontrarmos a complexidade da última função.

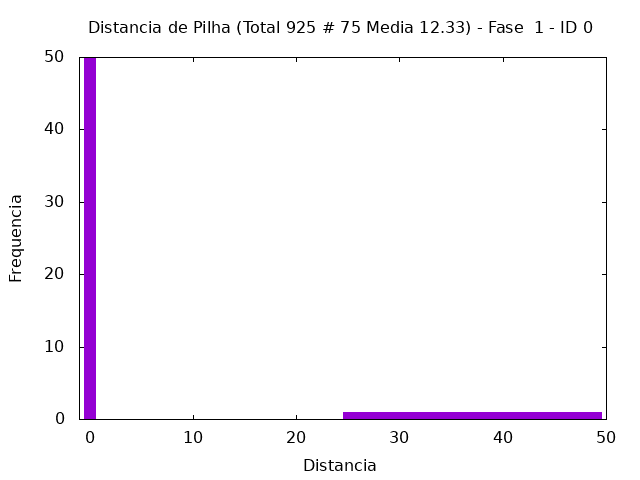
# *transpoeMatrizes*: Essa função realiza todas as operações considerando estruturas auxiliares unitárias . Ademais, uma matriz de tamanho é utilizada, acessando todos os seus elementos. Dessa forma, teremos a complexidade espacial da função como:

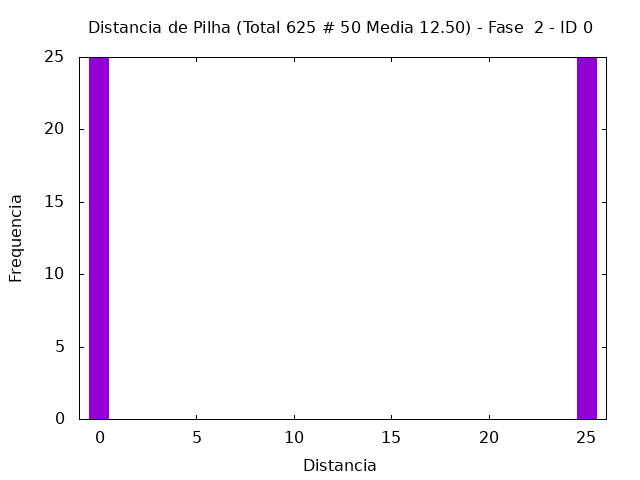
Para realizar a análise da complexidade de espaço do procedimento de transposição de matrizes, utilizaremos apenas uma matriz de valores aleatórios de dimensão 5x5. Para facilitar o entendimento, a execução do Makefile gera gráficos de acesso, distâncias de pilha e evolução dessas distâncias, utilizando as bibliotecas *gprof* e *analisamem*.









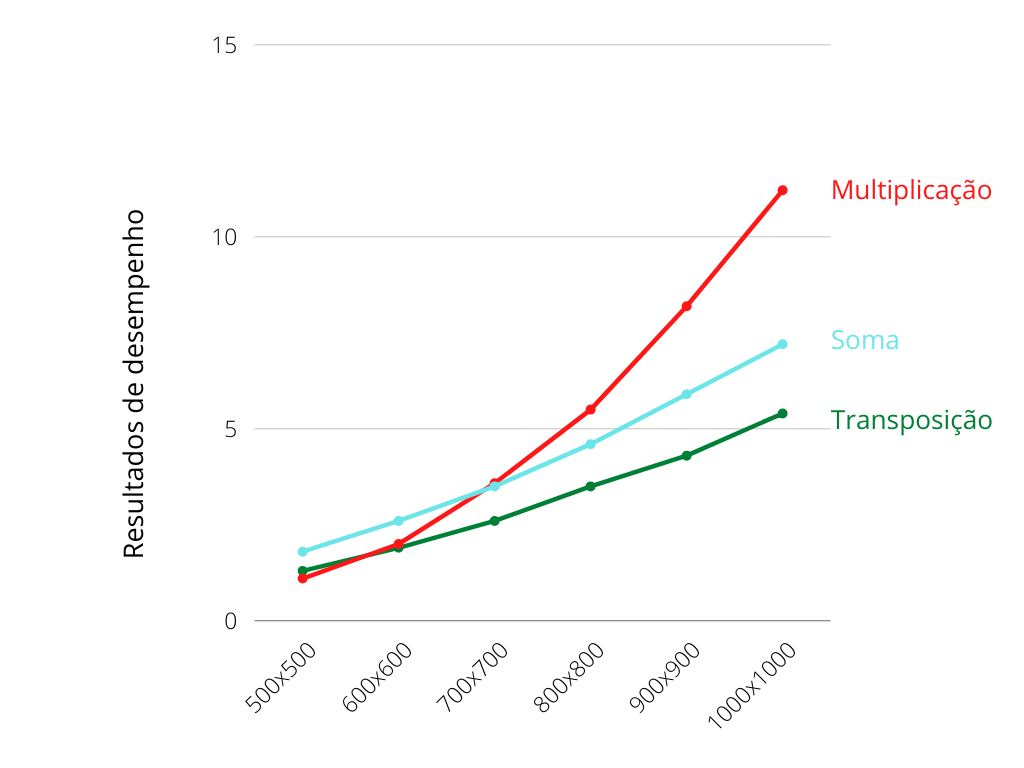


Analisando primeiramente o gráfico de acesso, percebemos que a matriz é inicializada com a função criaMatrizTxt, que lê um arquivo e atribui todos os valores da matriz. Em seguida, a matriz é acessada pela função acessaMatriz e a função transpoeMatriz também é chamada. Nessa função, uma matriz auxiliar é criada para transpor os elementos da matriz A, que são acessados linha a linha, formando a linha roxa nos endereços superiores. Dessa forma, quando os valores de A são lidos linha a linha, eles são atribuídos coluna a coluna na matriz auxiliar, que possui um endereço diferente, por isso a distância dos endereços no gráfico. Por fim, a matriz A recebe o endereço dos atributos da matriz auxiliar, que passa pela função *acessaMatriz* e *escreveArquivoTxt*.

Além disso, podemos interpretar os gráficos de distância e evolução de pilha, para entender melhor o funcionamento desse procedimento de transposição. Na primeira fase, ocorre um comportamento semelhante às outras funções, que é a inicialização da matriz, em que a distância de pilha é zero, já que os endereços nunca haviam sido acessados. Em seguida, na segunda fase, a distância de pilha aumenta linearmente até chegar em 50, já que todos os elementos da matriz auxiliar ainda não foram acessados, sua memória é adicionada a pilha, gerando uma distância de pilha zero, mas quando um elemento da matriz original A é acessado novamente, a distância de pilha aumenta cada vez mais, já que agora existe mais 25 elementos na pilha. Por fim, na última fase ocorre um processo muito semelhante ao procedimento das operações de soma e multiplicação, que é a chamada das funções acessaMatriz e escreveArquivoTxt, que gera as distâncias de pilha 0 e 25.

**CONCLUSÃO – Análise de complexidade**

Podemos concluir, portanto, que entre os três procedimentos utilizados no programa, (soma, multiplicação e transposição), o processo que mais demanda tempo de execução é a multiplicação, assim como o que tem a maior complexidade de tempo. Enquanto os outros dois processos apresentam complexidade assintótica igual a , a multiplicação apresenta . Para deixar isso mais evidente, podemos observar o seguinte gráfico, que compara o crescimento dos resultados das três operações. Porém, para realizar a construção desse gráfico, os resultados de desempenho de soma e transposição foram multiplicados por 10, já que os números de multiplicação são muito maiores.



# Conclusão

Este trabalho lidou com o problema de implementação de um tipo abstrato de dados de matrizes, em que suas dimensões são alocadas dinamicamente, o que nos permite lidar com dimensões arbitrariamente grandes. Nesse sentido, a abordagem utilizada foi a criação de um programa na linguagem C, que utiliza uma estrutura chamada *matriz,* que implementa seus atributos. Também foram implementadas algumas funções para melhorar a legibilidade e o reuso do código, como por exemplo funções que somam, multiplicam e transpõe matrizes.

Com a solução adotada, pode-se verificar que é possível realizar operações com matrizes de dimensões arbitrariamente grandes, como foi exemplificado durante os testes de performance (seção 4), em que foram utilizadas matrizes de tamanho 1000x1000, ou seja, 1 milhão de elementos. Essa solução permitiu realizar uma análise sucinta de como cada operação funciona na memória, tanto em relação ao tempo como em relação ao espaço, observando os gráficos e resultados de desempenho gerados pela execução do Makefile juntamente com as bibliotecas *analisamem* e *gprof*.

Por meio da resolução desse trabalho, foi possível praticar os conceitos relacionados aos Tipos Abstratos de Dados implementados na linguagem C e melhor entendimento a respeito dos arquivos Makefile. Além disso, durante a implementação da solução para o problema, um dos grandes desafios foi realizar a análise de complexidade espacial e temporal, que ao início do trabalho não estava muito familiarizado, mas o trabalho prático facilitou muito a compreensão.

# Referências

Slides virtuais e códigos da disciplina de estruturas de dados. Disponibilizado via *moodle*. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.