Trabalho Prático 3

Raissa Gonçalves Diniz
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG – Brasil
raissagdiniz@gmail.com

1. Introdução:

O presente trabalho se debruça sobre um problema peculiar, onde o lúdico e o matemático se entrelaçam em uma brincadeira de transformação de pontos no plano através da multiplicação de matrizes. Através de uma sequência de matrizes 2x2, em que cada uma representa uma transformação linear aplicada em um instante específico, foi-se investigada uma situação onde tais transformações afetam as coordenadas de pontos em uma folha de papel (sendo um ponto um vetor de dimensões 2x1).

Neste cenário, duas operações se mostraram fundamentais: a atualização, que permite a substituição de uma matriz em um dado instante, e a consulta, que visa determinar as coordenadas finais de um ponto dado seus instantes de "nascimento" e "morte". O objetivo final da brincadeira é determinar a posição em que o ponto morre, uma vez que você sabe o tempo e a posição onde ele nasce, o tempo em que ele morre e as transformações que serão aplicadas a ele. A complexidade reside em executar estas operações de maneira eficiente, o que nos leva à adoção de uma estrutura de dados robusta e adaptada ao problema: a árvore de segmentação, ou segtree.

A árvore de segmentação é uma estrutura de dados que permite pré-computar soluções para subarranjos específicos, otimizando tanto consultas quanto atualizações. Neste trabalho, a implementação desta estrutura de dados foi explorada no contexto do problema proposto, de forma a demonstrar sua eficácia e eficiência na execução dessa brincadeira.

2. Método

- Programa desenvolvido em C++
- Compilador g++ versão 9.4.0
- Windows 10 Home x64
- Linux Ubuntu 20.04

Especificações da máquina utilizada:

- Processador: Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz 2.40 GHz
- RAM: 8 GB

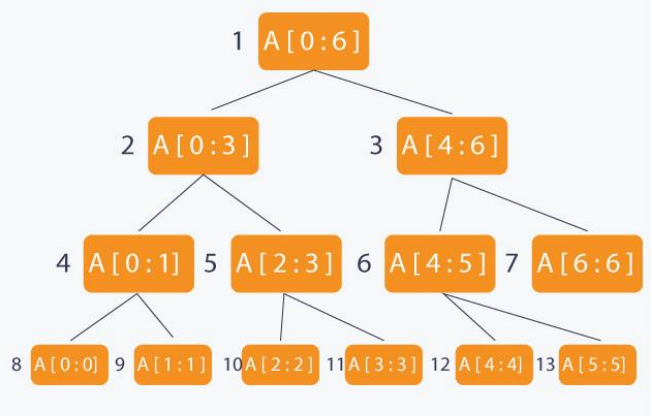
2.1 Estrutura de dados

A escolha da árvore de segmentação (segtree) como estrutura de dados para o problema em questão foi motivada por sua capacidade de lidar com as exigências específicas do cenário proposto: a necessidade de atualizações frequentes e a manipulação de números grandes, demandando rapidez nas operações matemáticas. A segtree é notoriamente

eficiente para problemas que envolvem intervalos ou segmentos, especialmente quando estes requerem alterações e consultas dinâmicas.

A atualização frequente das matrizes é um desafio significativo, pois cada atualização pode potencialmente afetar todas as consultas subsequentes. A segtree, com sua estrutura hierárquica, permite atualizações em tempo logarítmico, uma vez que atualizações em um nó podem ser propagadas eficientemente ao longo da árvore. Isso é crucial para manter a agilidade do sistema mesmo quando lidamos com uma grande quantidade de atualizações.

Quanto aos números grandes, a segtree permite a agregação de resultados pré-computados de forma que as operações matemáticas necessárias para responder a uma consulta possam ser realizadas em uma fração do tempo que levariam se fossem recalculadas do zero a cada vez. Isso é possível porque a árvore de segmentação divide o conjunto de dados em segmentos menores que podem ser rapidamente combinados para formar a solução de um intervalo maior.



A eficiência em consultas vem da habilidade da segtree de dividir o problema em partes menores e combinar os resultados parciais para obter a resposta final. Isso significa que, para determinar as coordenadas finais de um ponto, a estrutura utiliza resultados já computados dos subarranjos relevantes, ao invés de realizar múltiplas operações matriciais a partir do início.

Portanto, ao aplicar a segtree ao problema proposto, se garante uma solução que não só atende aos requisitos de atualizações e consultas eficientes mas também é escalável e robusta frente ao aumento do tamanho dos dados e da complexidade das operações, tornando-se uma escolha estratégica para o cenário descrito.

2.2 Funções e classes

No total, o trabalho contou com 3 classes*: IntArray, Matrix, e SegTree.* Segue a descrição de cada uma delas:

Classe intArray:

A classe *IntArray* é uma implementação personalizada de um vetor que armazena elementos do tipo *unsigned long long* (devido a natureza dos enormes números

multiplicados durante as contas, evitando overflow). Segue um resumo de cada um dos seus métodos e operadores sobrecarregados:

- *IntArray():* Construtor padrão que inicializa um vetor vazio.
- *~IntArray():* Destrutor que libera a memória alocada pelo vetor.
- *IntArray(const IntArray& other):* Construtor de cópia que cria um vetor como cópia de outro.
- *operator[](size_t index) const*: Operador de acesso para leitura que retorna um elemento em um índice especificado.
- *push_back(unsigned long long element):* Adiciona um novo elemento ao final do vetor, aumentando a capacidade do vetor se necessário.
- *operator[](size_t index):* Operador de acesso para modificação que retorna uma referência a um elemento em um índice especificado, permitindo alterá-lo.
- *size() const:* Retorna o tamanho atual do vetor (quantidade de elementos).
- clear(): Limpa o vetor, removendo todos os elementos e resetando sua capacidade e tamanho.
- *print() const:* Imprime todos os elementos do vetor.
- *operator=(const IntArray& other):* Sobrecarga do operador de atribuição para copiar elementos de outro vetor para o atual.

Classe Matrix:

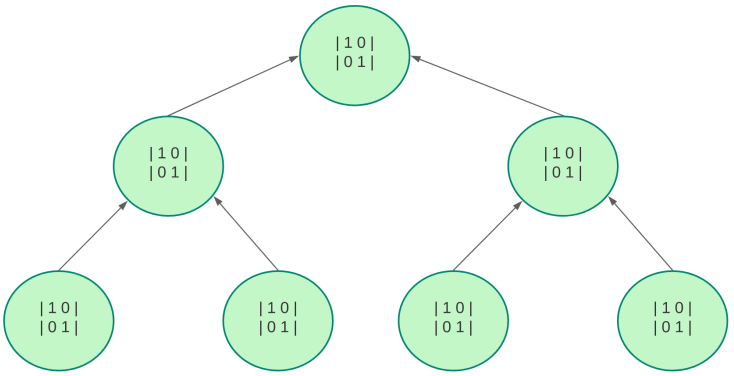
A classe *Matrix* é uma implementação personalizada de uma matriz 2x2 que oferece métodos para manipulação de matrizes nesse formato específico. Aqui está um resumo de cada um dos métodos:

- *Matrix():* Construtor padrão que inicializa a matriz como uma matriz identidade 2x2.
- *Matrix(unsigned long long a, unsigned long long b, unsigned long long c, unsigned long long d):* Construtor que inicializa a matriz com valores específicos dados para cada posição.
- *set(int row, int col, unsigned long long value):* Define o valor em uma posição específica da matriz.
- *get(int row, int col) const:* Obtém o valor em uma posição específica da matriz.
- *multiplyBy2x2(const Matrix& other):* Multiplica a matriz atual por outra matriz 2x2 e retorna o resultado, retornando apenas os 8 algoritmos menos significativos para prevenir overflow.
- *multiplyBy2x1(const IntArray& mat2x1):* Multiplica a matriz por um vetor coluna 2x1 e retorna o resultado em um objeto IntArray, também retornando apenas os 8 algoritmos menos significativos.
- *isIdentity() const:* Verifica se a matriz atual é uma matriz identidade.
- *Matrix(const Matrix& other):* Construtor de cópia que inicializa uma matriz com os mesmos valores de outra matriz 2x2.
- *operator=(const Matrix& other):* Sobrecarga do operador de atribuição para copiar os valores de outra matriz para a matriz atual.

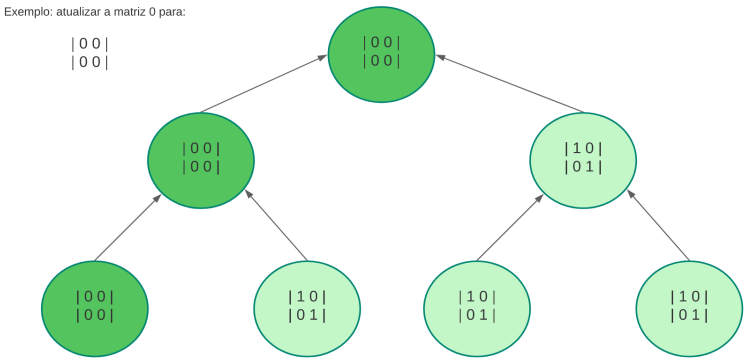
Classe SegTree:

A classe *SegTree* é uma implementação personalizada de uma árvore de segmentos para matrizes 2x2. Os métodos principais são:

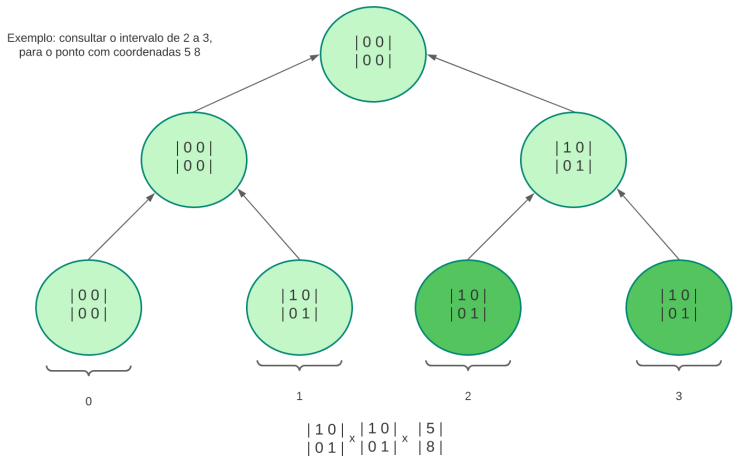
- *SegTree(int n):* Construtor que inicializa a árvore de segmentos com matrizes identidade com base no tamanho dado, ajustando-se à próxima potência de dois.



- *~SegTree():* Destrutor que libera a memória alocada pela árvore.
- *updateTree(int node, int start, int end, int idx, const Matrix& val):* Atualiza um nó específico na árvore de segmentos com uma nova matriz.



- *queryTree(int node, int start, int end, int l, int r) const:* Consulta a árvore de segmentos em um intervalo específico e retorna o produto das matrizes desse intervalo.



- *update(int idx, const Matrix& val):* Método público para atualizar uma das matrizes na árvore de segmentos.

- *query(int l, int r) const:* Método público para consultar um intervalo na árvore de segmentos e retornar o produto das matrizes nesse intervalo.

Integração:

Quando um ponto nasce, a classe *SegTree* pode ser consultada para encontrar o produto das matrizes aplicadas desde o nascimento até a morte do ponto, usando o método *query*. Se uma transformação muda em um certo ponto, a classe *SegTree* atualiza sua árvore com a nova matriz usando o método *update*. O resultado do produto das matrizes é então aplicado ao vetor coluna inicial que representa o ponto, usando o método *multiplyBy2x1* da classe *Matrix*.

Portanto, *IntArray* fornece a estrutura de dados para os pontos, *Matrix* realiza as operações matemáticas necessárias, e *SegTree* otimiza o processamento das operações de atualização e consulta. Juntas, estas classes formam um sistema eficiente para o problema de transformar pontos no plano com o uso de matrizes e otimizar o cálculo das coordenadas finais de um ponto após uma série de transformações.

3. Análise de complexidade (tempo e espaço):

Classe IntArray:

- `*IntArray()`:* Tempo O(1), Espaço O(1). A inicialização de um vetor vazio é uma operação constante, pois não envolve alocação de memória além da própria estrutura.

- `*~IntArray()*`: Tempo O(1), Espaço O(1). A liberação de memória é uma operação de tempo constante, independente do tamanho do vetor.

- `*IntArray(const IntArray& other)`:* Tempo O(1), Espaço O(1). A cópia de um vetor requer a alocação de um novo espaço de memória e a cópia de cada elemento individualmente. Contudo, como a classe é utilizada apenas para representar vetores de tamanho 2, o tempo e espaço são constantes.

- `*operator[](size_t index) const`:* Tempo O(1), Espaço O(1). O acesso a um elemento é direto e não requer memória adicional.

- `*push_back(unsigned long long element)`:* Tempo O(1), Espaço O(1). Só inserir o elemento no final do vetor. Nesse caso, como o vetor sempre tem tamanho 2, realocações não são necessárias.

- `*operator[](size_t index)`:* Tempo O(1), Espaço O(1). Similar ao operador de acesso constante, permite modificar um elemento existente.

- `*size() const`: Tempo O(1), Espaço O(1).* Retorna uma propriedade da classe sem cálculo adicional.

- `*clear()`: Tempo O(1), Espaço O(1).* Libera a memória e reinicia as variáveis, independente do tamanho do vetor.

- `*print() const*`:* Tempo O(1), Espaço O(1). Nesse caso, a classe sempre imprimirá dois elementos.

- `*operator=(const IntArray& other)*`:* Tempo O(1), Espaço O(1). Assim como no construtor de cópia, como o tamanho é sempre 2, isso seria O(1) para tempo e espaço na prática.

Classe Matrix:

- `*Matrix()*`:* Tempo O(1), Espaço O(1). Inicializar uma matriz identidade é uma operação constante, pois sempre terá tamanho 2x2.
- `*Matrix(unsigned long long a, unsigned long long b, unsigned long long c, unsigned long long d)*`:* Tempo O(1), Espaço O(1). Inicializa a matriz com valores específicos, o que é uma operação de tempo constante.
- `*set(int row, int col, unsigned long long value)*`:* Tempo O(1), Espaço O(1). Definir um valor em uma posição específica é uma operação de tempo constante.
- `*get(int row, int col) const*`:* Tempo O(1), Espaço O(1). Obter um valor de uma posição específica também é uma operação de tempo constante.
- `*multiplyBy2x2(const Matrix& other)*`:* Tempo O(1), Espaço O(1). Multiplica duas matrizes 2x2, o que envolve um número constante de operações aritméticas.
- `*multiplyBy2x1(const IntArray& mat2x1)*`:* Tempo O(1), Espaço O(1). Multiplica a matriz 2x2 por um vetor 2x1, também envolvendo um número constante de operações.
- `*isIdentity() const*`:* Tempo O(1), Espaço O(1). Verifica se a matriz é a identidade, o que requer a checagem de quatro valores fixos.
- `*Matrix(const Matrix& other)*`:* Tempo O(1), Espaço O(1). O construtor de cópia para uma matriz 2x2 envolve copiar um número fixo de elementos.
- `*operator=(const Matrix& other)*`:* Tempo O(1), Espaço O(1). A atribuição de uma matriz 2x2 envolve copiar um número fixo de elementos.

Classe SegTree:

- `*SegTree(int n)*`:* Tempo O(n), Espaço O(n). O construtor aloca espaço para uma árvore de segmentos completa baseada na próxima potência de dois do tamanho do input, inicializando todos os nós como matrizes identidade.
- `*~SegTree()*`:* Tempo O(1), Espaço O(1). O destrutor simplesmente libera a memória alocada para a árvore, que é uma operação de tempo constante.
- `*updateTree(int node, int start, int end, int idx, const Matrix& val)*`:* Tempo O(log n), Espaço O(log n). A atualização de um nó na árvore de segmentos é uma operação logarítmica porque potencialmente envolve percorrer a altura da árvore, que é logarítmica em relação ao número de elementos.
- `*queryTree(int node, int start, int end, int l, int r) const*`:* Tempo O(log n), Espaço O(log n). A consulta por um intervalo requer percorrer a árvore até a profundidade logarítmica para encontrar e combinar matrizes, envolvendo chamadas de pilha recursivas.
- `*update(int idx, const Matrix& val)*`:* Tempo O(log n), Espaço O(log n). Este método é um wrapper público para `updateTree`, portanto, mantém a mesma complexidade ao iniciar no nó raiz.
- `*query(int l, int r) const*`: Tempo O(log n), Espaço O(log n). Similar a `update`, este método é um wrapper público para `queryTree` e mantém a mesma complexidade.

4. Estratégias de robustez:

Várias estratégias de robustez foram empregadas para garantir a integridade e a correta utilização das classes *`IntArray`, `Matrix`,* e *`SegTree`*. Aqui estão elas:

Classe IntArray:

1. O destrutor *(`~IntArray`)* e os métodos *`clear`* e *`operator=`* garantem que a memória alocada seja liberada corretamente, prevenindo vazamentos de memória.

2. Os operadores de acesso *(`operator[]`*) lançam uma exceção *`std::out_of_range`* se um índice inválido for acessado, evitando assim acessos fora dos limites do array.

3.O construtor de cópia e o operador de atribuição criam cópias profundas dos objetos *`IntArray`*, assegurando que mudanças em cópias não afetem o objeto original.

Classe `Matrix:

1.Os métodos *`set`* e *`get`* lançam exceções *`std::out_of_range`* se índices fora dos limites da matriz 2x2 forem utilizados, garantindo acessos seguros aos elementos da matriz.

2. O método *`multiplyBy2x1`* lança uma exceção `std::invalid_argument` se o tamanho do vetor fornecido para multiplicação for inválido, o que assegura que as operações de multiplicação sejam realizadas corretamente.

3. Durante a multiplicação de matrizes e a multiplicação de matrizes por vetores, o código aplica o módulo (retorna os 8 algoritmos menos significativos) a cada etapa para prevenir o overflow de `unsigned long long`.

4. A classe *`Matrix`* fornece múltiplos construtores para inicializar matrizes de diferentes maneiras, incluindo a matriz identidade e matrizes com valores específicos.

Classe SegTree:

1. No construtor, a árvore de segmentos é inicializada com matrizes identidade para garantir que todos os nós estejam em um estado válido e cumprindo as especificações do enunciado.

2. O destrutor *(`~SegTree`)* libera a memória alocada para a árvore de segmentos, evitando vazamentos de memória.

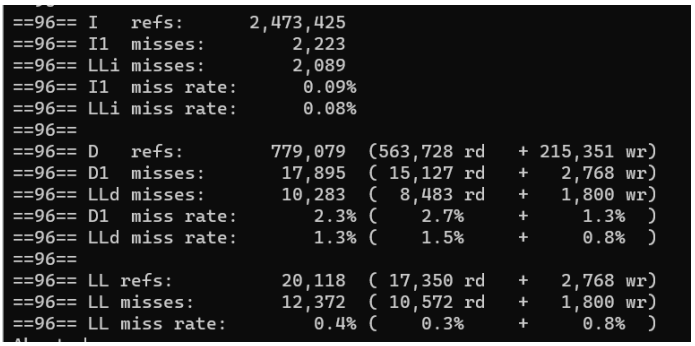
3. O método *`updateTree`* atualiza o valor de um nó e seus nós pai de forma recursiva, garantindo que a estrutura da árvore permaneça consistente após a atualização.

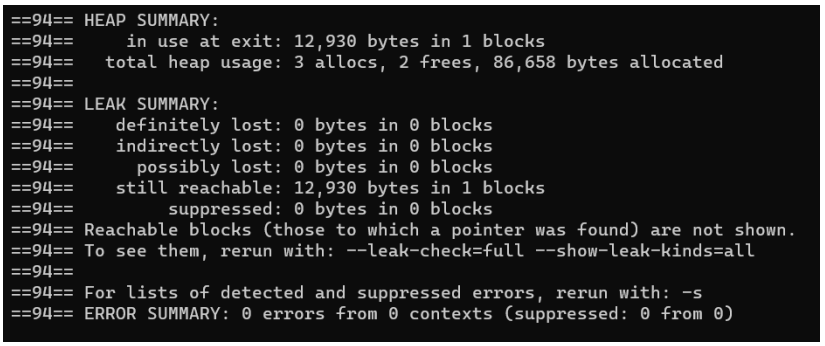
4. O método *`queryTree`* implementa checagens para garantir que apenas intervalos válidos sejam consultados e combina resultados de consultas parciais de forma correta.

5. Análise experimental:

Inicialmente, a análise experimental foi utilizada para avaliar o uso de memória durante a execução do programa, permitindo identificar possíveis vazamentos de memória e otimizar a alocação e desalocação de memória pelo programa. No caso do código apresentado, foi utilizado o Valgrind para realizar essa análise, uma ferramenta que permite identificar vazamentos de memória. Para utilizar o Valgrind, basta executar o programa com o comando "valgrind nome_do_programa". O Valgrind irá monitorar o uso de memória pelo programa e exibir possíveis erros de vazamento de memória. Ele foi executado com as opções *"--leak-check=full"* e *"--tool=cachegrind"* para realizar uma verificação completa de vazamentos de memória. A versão específica utilizada foi a 3.15.0.

Após a execução do programa, o Valgrind gerou um resumo mostrando que não há erros de memória detectados no que diz respeito a vazamentos ("definitely lost", "indirectly lost", "possibly lost" estão todos com 0 bytes), o que é positivo. Os dados de referência e miss rate de cache fornecidos pelo Cachegrind mostram uma taxa de erros muito baixa para o cache de instruções (I1 e LLi) e uma taxa de erro um pouco maior, mas ainda assim baixa, para o cache de dados (D1 e LLd). Isso implica que o programa foi geralmente eficiente em termos de acesso ao cache, com a maioria dos acessos ao cache resultando em hits e não misses, como mostrado nas imagens a seguir:





Após esse experimento inicial, foi realizada uma análise para comparar o desempenho da árvore de segmentação em relação a uma abordagem mais simples para resolução desse problema: armazenar cada uma das transformações em um arranjo indexado pelos instantes de tempo. Apenas para esse fim, um vetor de matrizes simples foi criado, com os principais métodos de consulta, atualização e inicialização.

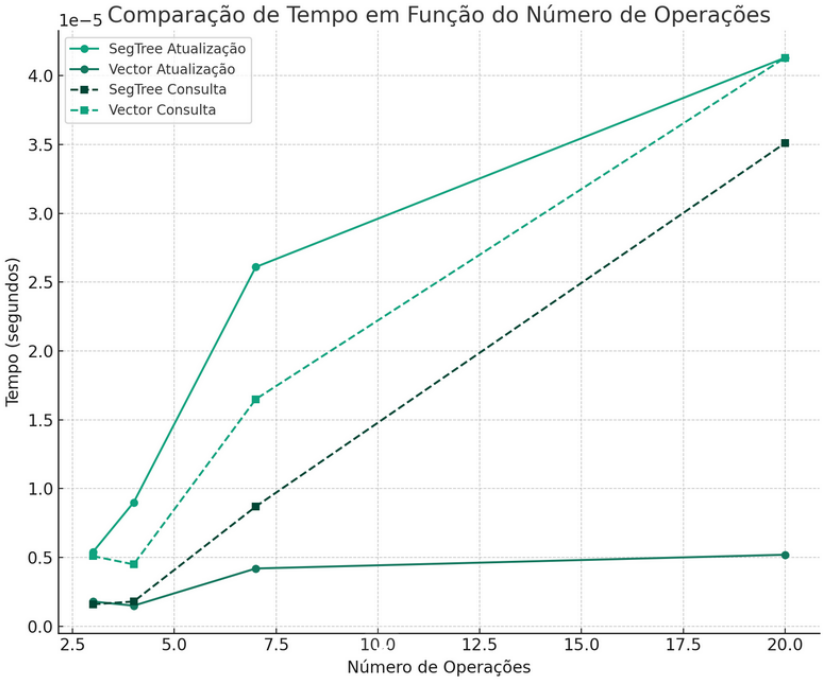
Para isso, um código de medição de tempo foi inserido manualmente em torno das seções do código desejadas, utilizando a biblioteca "*std::chrono"*, que é uma biblioteca de data e hora do C++ Standard Library. O código foi instrumentado em torno de cada operação de atualização e consulta, envolvendo cada chamada de função individual em suas próprias medições de tempo e acumulando esses tempos separadamente, tal qual o exemplo de código a seguir. Entradas com várias operações de atualização e consulta foram passadas para ambas estruturas de dados.

```
using namespace std::chrono;
…

duration<double> total_update_time(0);
duration<double> total_update_time(0)
…
```

```
auto start = high_resolution_clock::now();
// operação de update
auto end = high_resolution_clock::now();
total_update_time += end - start;
…
auto start = high_resolution_clock::now();
// operação de consulta
auto end = high_resolution_clock::now();
total_query_time += end - start;
…
```

Com os dados já coletados, a biblioteca Matplotlib, da linguagem Python, foi utilizada para a geração dos gráficos. Caso deseje testá-la, você precisa ter tanto o Python instalado, como a biblioteca (abra um terminal ou prompt de comando e execute o seguinte comando: *"pip install matplotlib"*). Um código em python identificando como a geração dos gráficos deve ser feita foi criado, e executado com o comando "*python graph_plot.py".* Os resultados são apresentados a seguir.



A análise dos dados representados nos gráficos sugere um padrão distinto em relação ao desempenho das estruturas de dados SegTree e Vector com o aumento do número de operações. Inicialmente, para um número pequeno de matrizes e operações, as diferenças de tempo entre as duas estruturas são mínimas. De fato, o Vetor mostra um tempo de atualização ligeiramente melhor do que a SegTree, o que pode ser atribuído à sua simplicidade e menor custo de operação inicial.

No entanto, à medida que o número de operações cresce, a SegTree começa a demonstrar sua superioridade em termos de eficiência de tempo, especialmente nas operações de consulta. A razão para isso reside na natureza das operações que cada estrutura executa. O Vector executa atualizações em tempo constante, o que é evidenciado pelos tempos de

atualização relativamente estáveis nos gráficos, mas suas consultas requerem uma operação linear para percorrer e multiplicar matrizes sequencialmente, resultando em um aumento significativo no tempo de consulta à medida que o número de operações cresce.

Por outro lado, a SegTree, uma estrutura de dados mais complexa e especializada, é projetada para otimizar tanto atualizações quanto consultas, aproveitando suas propriedades de árvore para executar operações em tempo logarítmico em relação ao número de elementos. Isso é particularmente eficaz para consultas, pois a SegTree pode combinar resultados de subárvores rapidamente, sem a necessidade de processar cada elemento individualmente.

À medida que o número de operações aumenta, o custo de tempo das operações de consulta na SegTree cresce muito mais lentamente do que no Vector, indicando uma escalabilidade muito melhor. Isso é crucial em aplicações onde o número de operações é grande e as consultas são frequentes, como em sistemas de processamento de transações ou em simulações computacionais complexas. Portanto, para cenários de uso que envolvem um grande volume de atualizações e consultas, a SegTree é uma escolha muito mais eficiente em comparação com o uso de um simples vetor de matrizes.

6. Conclusões:

No decorrer deste trabalho prático, tive a oportunidade de aprofundar meus conhecimentos e habilidades na implementação de estruturas de dados complexas e seus respectivos algoritmos de manipulação, ficando cara a cara com problemas de gestão eficiente de tempo e memória, de maneira a garantir operações seguras e confiáveis. As estratégias de robustez adotadas, como gerenciamento de recursos, validação de índices e prevenção de overflow, não apenas reforçaram a estabilidade e a confiabilidade do código, mas também me proporcionaram insights valiosos sobre as melhores práticas de programação. Através da construção dessas classes e da implementação de suas funcionalidades, solidifiquei minha compreensão sobre a importância da escrita de código seguro e eficiente, além de refinar minhas habilidades em raciocínio lógico e análise de algoritmos.

7. Referências:

Aula 9 - SegTree. Maratona UFMG, , 23 de jan de 2021. Disponível em: <https://youtu.be/OW_nQN-UQhA?si=Br2fa3HyO_nTNmRS>

Sum of given range | Segment tree construction and update | Simplest explanation. Techdose, , 8 de fev. de 2020. Disponível em: <https://www.youtube.com/watch?v=2bSS8rtFym4&t=1288s&ab_channel=Techdose>

Segment Tree and Lazy Propagation. , [s.d.]. Disponível em: <https://www.hackerearth.com/practice/notes/segment-tree-and-lazy-propagation/>

Todas as imagens de exemplificação do funcionamento do programa são de autoria própria, e foram criadas usando a ferramenta https://lucid.app.