UNIVERSIDADE FEDERAL DE MINAS GERAIS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

LEAN HENRIQUE PEREIRA MIRANDA
2022035652

TRABALHO PRÁTICO 03

BELO HORIZONTE
2023

Introdução:

Neste trabalho, abordamos a relação entre transformações lineares e matrizes para lidar com uma sequência de pontos em um plano bidimensional. Cada ponto é inicialmente associado à uma matriz identidade. As crianças podem realizar duas operações: atualização, substituindo uma matriz por outra em um instante específico, e consulta, para determinar as coordenadas finais de um ponto.

Para resolver esse desafio, optamos por uma abordagem eficiente: a utilização de uma árvore de segmentação. Essa estrutura permite gerenciar e processar as operações de forma otimizada, tornando a análise das transformações ao longo do tempo mais simples e rápida.

Ao explorar a interseção entre conceitos matriciais e computacionais, demonstramos como técnicas matemáticas podem ser aplicadas de maneira prática e direta na solução de problemas computacionais do mundo real.

Método:

No decorrer deste projeto, adotei uma estrutura de dados fundamental: a árvore de segmentação, que desempenhou um papel central na resolução do problema proposto. Além disso, dois Tipos Abstratos de Dados (TADs) foram essenciais para a implementação: o TipoNo e a matriz.

O TipoNo foi concebido para armazenar as variáveis cruciais da árvore. Cada nó representa um intervalo e mantinha uma matriz associada, além de ponteiros para os filhos esquerdo e direito. Essa abordagem permitiu uma construção dinâmica e eficiente da estrutura da árvore.

A matriz, por sua vez, foi projetada especificamente para representar matrizes 2x2. Essa escolha simplificou a manipulação dos dados, tornando a implementação mais acessível e intuitiva. Cada instância da matriz continha um array de inteiros 2x2 representando os elementos da matriz.

A Árvore de Segmentação foi o alicerce que conectou esses conceitos, proporcionando uma representação hierárquica dos intervalos e permitindo a aplicação das operações de atualização e consulta de maneira eficiente. Em resumo, a combinação desses elementos ofereceu uma solução robusta e elegante para a resolução do problema proposto.

Funções:

A solução para o problema em questão foi implementada através de três funções centrais, cada uma desempenhando um papel fundamental. A função ConstróiArvore é responsável por receber a quantidade de instâncias de tempo fornecidas pelo usuário e construir a árvore, dividindo os intervalos com base nessa quantidade. A função atualizar é acionada quando uma instância de tempo específica e uma matriz 2x2 são fornecidas, alterando a matriz do nó correspondente na árvore.

A terceira função crucial é a consulta, que recebe um intervalo, além de dois pontos, x e y, e realiza a multiplicação desses pontos pela matriz associada ao nó que representa o intervalo fornecido. Vale ressaltar duas funções secundárias, mas igualmente significativas: CalculaResultado, responsável por calcular a matriz de um nó específico com base na multiplicação de seus filhos, e multiplicaMatrizes, que realiza a multiplicação de matrizes, considerando apenas os 8 dígitos menos significativos para lidar com números muito grandes.

Análise de complexidade:

Segue abaixo a análise de complexidade das 3 principais funções desenvolvidas no algoritmo.

ConstróiArvore :

A complexidade de tempo da função `ConstroiArvore` é O(n), onde "n" é o número de instâncias de tempo representadas na árvore.

1. A condição `if (p == NULL)` implica em uma única alocação de memória para o nó quando o mesmo ainda não existe. Essa operação é realizada em tempo constante, O(1).
2. A condição `if (inicio == fim)` é alcançada quando um nó folha é atingido. Aqui, é gerada uma matriz identidade, e essa operação também é realizada em tempo constante, O(1).
3. O restante da função é recursivo, dividindo o intervalo ao meio e chamando a função para os filhos esquerdo e direito. A função é chamada uma vez para cada intervalo possível, cobrindo todos os nós da árvore. Portanto, a complexidade de tempo total é proporcional ao número total de nós, que é O(n).

   Portanto, a complexidade de tempo da função `ConstroiArvore` é linear em relação ao número de instâncias de tempo "n" representadas na árvore.

A complexidade de memória da função `ConstroiArvore` também é O(n), onde "n" é o número de instâncias de tempo representadas na árvore.

1. Cada nó da árvore consome uma quantidade constante de memória para armazenar a matriz 2x2, os dois inteiros representando o intervalo e os ponteiros para os filhos

esquerdo e direito. Essa quantidade constante não depende do número total de instâncias de tempo.
2. A função é chamada recursivamente para os filhos esquerdo e direito, o que implica em ativar uma nova chamada de função e alocar memória para os nós correspondentes. O número total de chamadas recursivas é proporcional ao número total de instâncias de tempo "n".

Portanto, a complexidade de memória é linear em relação ao número de instâncias de tempo representadas na árvore.

Atualizar:

A complexidade da função atualizar é O(logN), onde N é o número total de elementos na sequência representada pela árvore. Isso ocorre porque a árvore de segmentação divide a sequência em intervalos e, durante a atualização, apenas os intervalos afetados são percorridos, resultando em uma complexidade logarítmica em relação ao número total de elementos.

1. A função é recursiva e começa pela raiz da árvore.
2. Verifica se a instância a ser atualizada está fora do intervalo representado pelo nó atual da árvore. Se estiver fora, a função retorna sem fazer nada.
3. Se o nó atual representa um único elemento (caso base), atualiza o valor do elemento com a matriz A.
4. Se o nó atual representa um intervalo maior, divide o intervalo em dois e chama recursivamente a função nos filhos esquerdo e direito .

A razão pela qual a complexidade é O(logN) está relacionada à forma como a árvore de segmentação é construída. Cada nível da árvore representa uma divisão subsequente do intervalo inicial. Como a árvore de segmentação é uma árvore binária, o número de níveis é logarítmico em relação ao número total de elementos N.
Quando um elemento é atualizado, a função percorre o caminho da raiz até a folha correspondente ao intervalo que contém o elemento a ser atualizado. Como a árvore é balanceada, o caminho percorrido é logarítmico em relação ao número total de elementos na sequência.
Portanto, a complexidade de tempo da função atualizar é O(logN), o que a torna eficiente para operações de atualização em sequências grandes.

A complexidade de espaço (memória) de uma árvore de segmentação é O(NlogN), onde N é o número total de elementos na sequência.
Isso ocorre porque a árvore de segmentação armazena informações sobre cada intervalo em cada nó. Como a árvore é uma estrutura de árvore binária, ela tem O(logN) níveis. Em cada nível, há uma quantidade de nós proporcional ao número total de elementos N. Portanto, o espaço total usado pela árvore é O(NlogN).
É importante notar que esse é um fator importante a ser considerado ao trabalhar com árvores de segmentação, especialmente em casos onde a memória disponível é limitada. Se a eficiência de espaço for uma preocupação crítica, podem ser consideradas outras estruturas de dados mais eficientes em termos de espaço, dependendo dos requisitos específicos do problema.

Consulta :

A complexidade da função de consulta é O(logN), onde N é o número total de elementos na sequência representada pela árvore. A razão para essa complexidade está relacionada à estrutura de árvore de segmentação, que divide recursivamente o intervalo em cada nó. A altura da árvore é O(logN), já que ela é uma árvore binária. Cada nível da árvore representa uma divisão adicional do intervalo, e a consulta percorre a árvore de cima para baixo, reduzindo continuamente o intervalo de busca pela metade.

1. Se o nó atual é nulo ou está completamente fora do intervalo de consulta [inicio,fim][inicio,fim], a função retorna uma matriz vazia. Isso é feito para garantir que não realizemos consultas desnecessárias em partes da árvore que não têm relação com o intervalo desejado.
2. Se o nó atual representa exatamente o intervalo de consulta [inicio,fim][inicio,fim], calculamos o resultado para esse nó e retornamos a matriz associada a ele. Este é o caso em que encontramos exatamente o intervalo desejado.
3. Se o nó atual não corresponde exatamente ao intervalo de consulta, verificamos os filhos à esquerda e à direita.
4. Se a consulta se sobrepõe a ambos os filhos (condição *fim* > *p->esq->fim && inicio* < *p->dir->início*), consultamos recursivamente ambos os filhos e multiplicamos os resultados.
5. Se a consulta está totalmente à esquerda ou à direita, consultamos apenas o filho correspondente.

Portanto, a função de consulta é eficiente para intervalos em sequências grandes, graças à estrutura hierárquica da árvore de segmentação.

A complexidade de memória da função `consulta` que você forneceu é principalmente afetada pelo número de chamadas recursivas e pela quantidade de informação armazenada em cada nó. Vamos analisar:

1. *Número de Chamadas Recursivas:*
   - O número de chamadas recursivas depende da estrutura da árvore e da posição do intervalo de consulta em relação aos intervalos dos nós. Em uma árvore de segmentação balanceada, a altura da árvore é \(\log N\), onde \(N\) é o número total de elementos na sequência.
   - Cada chamada recursiva cria uma nova chamada na pilha de execução, contribuindo para a complexidade de memória.

2. *Quantidade de Informação por Chamada Recursiva:*
   - Cada chamada recursiva cria matrizes locais (`resultEsq` e `resultDir`) e outros dados locais, contribuindo para a complexidade de memória.
   - A quantidade de informação por chamada recursiva é constante e não depende do tamanho total da sequência.

Com base nisso, a complexidade de memória da função `consulta` é predominantemente afetada pelo número de chamadas recursivas, resultando em uma complexidade de

memória de O(log N), onde N é o número total de elementos na sequência representada pela árvore.

Análise de Robustez:

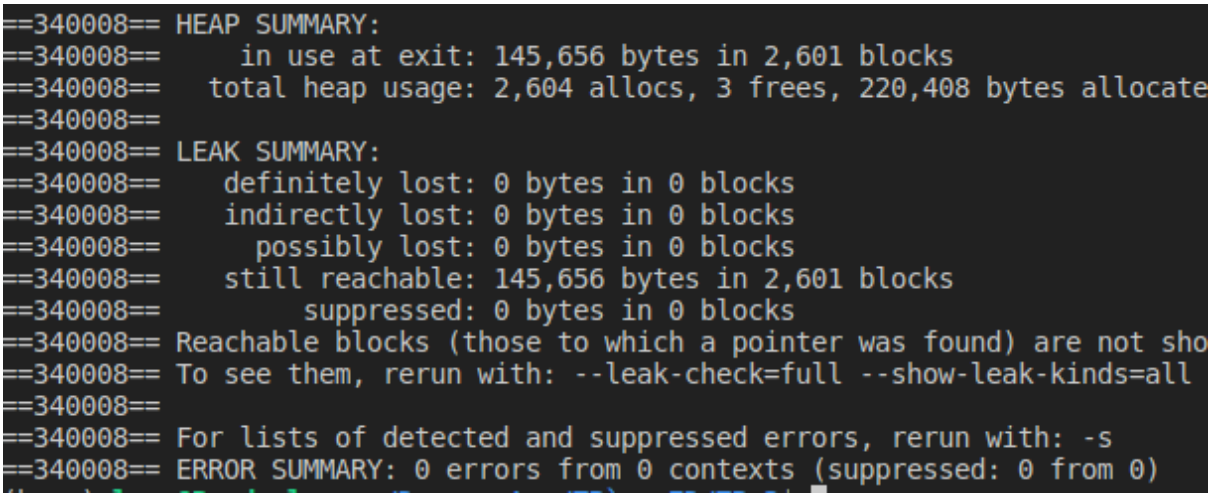
No projeto foram implementadas várias práticas de programação defensiva, tentei comentar o código o máximo possível para evitar dúvidas na interpretação, além disso procurei identar o código de maneira que facilitasse a leitura e não ficasse confuso. Também procurei modularizar o código de maneira eficiente para facilitar um entendimento e possíveis ajustes futuros.

Análise Experimental: Como experimento realizado em termos de desempenho computacional, usei o valgrind e o cachegrind mostrado em sala de aula para analisar os comportamentos de memória do projeto

.

VALGRIND

Segue abaixo o retorno das informações fornecidas pelo valgrind e a análise delas.



O relatório do Valgrind indica que não foram detectados vazamentos de memória definitivos (definitely lost), ou seja, o programa liberou toda a memória alocada antes de encerrar. No entanto, há ainda blocos de memória que são considerados "ainda acessíveis" (still reachable) no final da execução. Esses blocos referem-se a alocações que não foram liberadas, mas ainda são acessíveis pelo programa. Neste, isso não é um problema, pois as alocações foram feitas para a vida útil do programa e não precisam ser explicitamente liberadas.

CACHEGRIND

Para obter mais conhecimento em termos de cache do funcionamento do meu projeto, usei o valgrind, passando um teste com 15 instâncias e 15 operações e obtive os seguintes dados:

Métricas de Desempenho do Cachegrind:

1. **Instruções:**
    - **Refs (referências):** 12.569.278 instruções processadas.
    - **I1 misses:** 2.282 falhas na cache L1 de instruções.
    - **LLi misses:** 2.152 falhas na última camada da cache de instruções.
    - **I1 miss rate:** Taxa de falha na cache L1 de instruções de 0,02%.
    - **LLi miss rate:** Taxa de falha na última camada da cache de instruções de 0,02%.
2. **Dados:**
    - **Refs:** 6.046.330 referências a dados realizadas.
    - **D1 misses:** 72.069 falhas na cache L1 de dados.
    - **LLd misses:** 61.898 falhas na última camada da cache de dados.
    - **D1 miss rate:** Taxa de falha na cache L1 de dados de 1,2%.
    - **LLd miss rate:** Taxa de falha na última camada da cache de dados de 1,0%.
3. **Total:**
    - **LL refs:** 74.351 referências à última camada da cache.
    - **LL misses:** 64.050 falhas na última camada da cache.
    - **LL miss rate:** Taxa de falha total na última camada da cache de 0,3%.

Interpretação Geral:

- As taxas de falha na cache são baixas, indicando um bom desempenho geral.
- A cache L1 de instruções apresenta um desempenho notável, com uma taxa de falha muito baixa.
- O desempenho da cache de dados é bom, com oportunidades potenciais de otimização.
- O tempo perdido devido a falhas na última camada da cache é baixo, sugerindo eficiência geral do cache

<u>Conclusão</u>:

Em conclusão, este trabalho demonstra a eficácia da aplicação de conceitos matriciais e estruturas computacionais na resolução de problemas práticos. A representação de transformações lineares por meio de matrizes, associadas a pontos em um plano bidimensional, ofereceu uma abordagem intuitiva.
A escolha da árvore de segmentação revelou-se acertada para gerenciar as operações ao longo do tempo de forma otimizada. As funções principais, como ConstróiArvore, atualizar e

consulta, desempenharam papéis fundamentais na solução do problema, destacando a importância da combinação entre conhecimentos matemáticos e computacionais.

Assim, este trabalho não apenas apresenta uma solução eficiente, mas também evidencia a aplicabilidade direta de conceitos matemáticos na solução de desafios computacionais do mundo real. Essa interseção oferece uma perspectiva prática e elegante para abordar problemas complexos, destacando a sinergia entre a teoria matemática e a implementação computacional.

Bibliografia:

Monteiro, B. (2023). Slides virtuais da Maratona de Programação da UFMG. Universidade Federal de Minas Gerais, Belo Horizonte.