

Análise e Implementação de Algoritmos Greedy e de Divisão e Conquista: Detecção de Padrões Temporais e Multiplicação Otimizada de Matrizes

Caroline da Rocha de Lima¹

¹Escola Politécnica – Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
90619-900– Porto Alegre – RS – Brazil

caroline.lima01@edu.pucrs.br

Abstract. *This report presents the analysis, development, and implementation of two algorithms, applying optimization strategies through Greedy and Divide-and-Conquer techniques. The first problem focuses on detecting patterns in series, determining whether a given sequence of events is a subsequence of a larger one, preserving order but not necessarily continuity. An efficient algorithm was developed to detect such patterns, including execution time analysis. The second problem explores matrix multiplication beyond the traditional $\Theta(n^3)$ time, considering V. Strassen's work. A matrix multiplication algorithm based on Divide-and-Conquer was implemented, with a comparison between traditional and optimized methods. Both problems were addressed with an emphasis on efficiency and algorithmic clarity, validated through tests in Java.*

Resumo. *Este relatório apresenta a análise, o desenvolvimento e a implementação de dois algoritmos, aplicando estratégias de otimização por meio das técnicas de algoritmos Greedy e de Divisão e Conquista. O primeiro problema aborda a detecção de padrões em séries temporais de eventos, verificando se uma sequência de eventos é uma subsequência de outra maior, respeitando a ordem mas não necessariamente a continuidade. Um algoritmo foi desenvolvido para identificar tais padrões, com análise de tempo de execução. O segundo problema investiga a multiplicação de matrizes além do tempo tradicional $\Theta(n^3)$, considerando o trabalho de V. Strassen. Foi implementado um algoritmo de multiplicação baseado na técnica de Divisão e Conquista, com comparação entre métodos tradicionais e otimizados. Ambos os problemas foram resolvidos com foco em eficiência e clareza algorítmica, validados por testes em Java.*

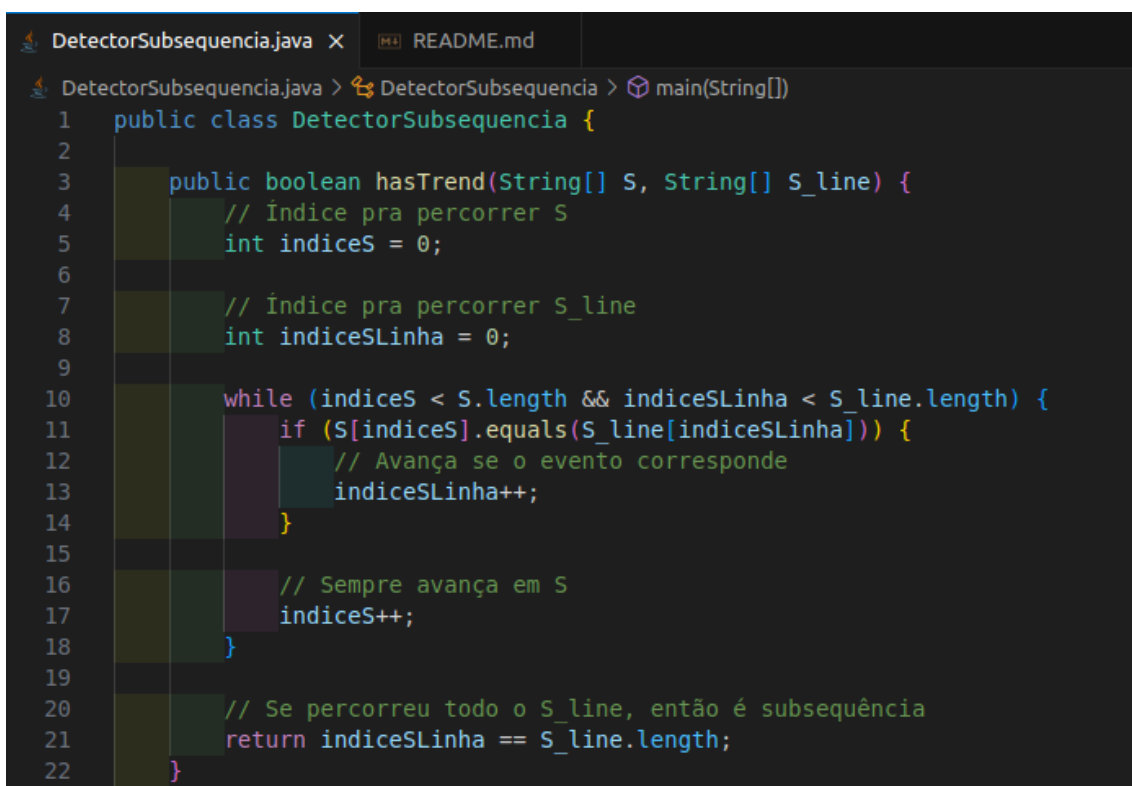
1. Problema 1

O primeiro problema proposto consiste em detectar padrões dentro de sequências temporais de eventos, como por exemplo, ordens de compra em bolsas de valores, que naturalmente seguem uma ordenação no tempo. Dado um conjunto de eventos e uma sequência longa S , o desafio é verificar se uma sequência menor S' ocorre dentro de S respeitando a ordem dos eventos, mesmo que não estejam consecutivos. Formalmente, o problema busca identificar se S' é uma subsequência de S . Este tipo de problema é relevante na mineração de dados temporais, onde a identificação de padrões frequentes e tendências é essencial para a análise preditiva e tomada de decisão.

No contexto da disciplina de Projeto e Otimização de Algoritmos, este problema permite aplicar técnicas de projeto eficiente de algoritmos, em especial os algoritmos greedy, que tomam decisões locais ótimas para buscar uma solução global eficiente. A escolha de um algoritmo greedy para esse problema é boa, pois ao percorrer a sequência S , podemos fazer decisões locais ao comparar cada evento com o próximo evento esperado de S' . Além disso, o problema envolve a análise da complexidade temporal da solução, algo fundamental na disciplina, que visa não apenas resolver problemas, mas também otimizá-los, garantindo que a solução seja escalável e adequada para grandes volumes de dados, como os encontrados em séries temporais reais.

2. Algoritmo Desenvolvido para o Problema 1

A implementação da solução para o problema de detecção de subsequência foi desenvolvida utilizando uma abordagem greedy e desenvolvida em Java, que se mostrou adequada para o problema. A lógica central do algoritmo baseia-se na ideia de percorrer a sequência S do início ao fim, comparando cada elemento com o elemento correspondente de S_line , a subsequência que se deseja verificar. Dois índices são utilizados: um para percorrer S e outro para percorrer S_line . A cada iteração, o algoritmo verifica se o evento atual de S corresponde ao evento atual de S_line . Se corresponder, o índice de S_line é incrementado, indicando que parte da subsequência foi encontrada. O índice de S é sempre incrementado, garantindo que os eventos sejam verificados na ordem correta, respeitando a definição de subsequência.



```
DetectorSubsequencia.java X README.md
DetectorSubsequencia.java > DetectorSubsequencia > main(String[])
1 public class DetectorSubsequencia {
2
3     public boolean hasTrend(String[] S, String[] S_line) {
4         // Índice pra percorrer S
5         int indices = 0;
6
7         // Índice pra percorrer S_line
8         int indiceSlinha = 0;
9
10        while (indices < S.length && indiceSlinha < S_line.length) {
11            if (S[indices].equals(S_line[indiceSlinha])) {
12                // Avança se o evento corresponde
13                indiceSlinha++;
14            }
15
16            // Sempre avança em S
17            indices++;
18        }
19
20        // Se percorreu todo o S_line, então é subsequência
21        return indiceSlinha == S_line.length;
22    }
```

A decisão de implementar desta forma se deu pela sua simplicidade e eficiência, uma vez que cada elemento de S é visitado no máximo uma vez, resultando em uma complexidade de tempo linear $O(n)$, onde n é o tamanho da sequência S . Essa

abordagem é eficiente para grandes volumes de dados, como validado pelos testes com sequências contendo até um milhão de elementos, apresentando bom desempenho mesmo em casos extremos. Além disso, a implementação foi validada com diversos testes, cobrindo cenários típicos e de borda, o que reforça sua robustez e eficiência na resolução do problema proposto.

3. Testes do Algoritmo 1 Desenvolvido

Para garantir a corretude e a eficiência do algoritmo implementado, foi desenvolvido alguns testes abrangentes, composta por 13 casos distintos. Esses testes foram elaborados para cobrir diferentes cenários possíveis, incluindo casos simples, casos limites e situações de grande volume de dados. Os primeiros testes validam subsequências presentes no início, meio e fim da sequência S, além de subsequências que não existem ou que possuem a ordem incorreta dos eventos. Foram também considerados casos especiais, como quando S_line está vazia (sempre considerada subsequência válida) e quando S está vazia, mas S_line não (sempre inválida).

Além disso, foram realizados testes de desempenho com sequências grandes, contendo até 1.000.000 de elementos, para avaliar o comportamento do algoritmo em situações de alto volume de dados, tanto em cenários favoráveis (onde a subsequência existe) quanto em casos extremos (onde a subsequência não está presente). Os tempos de execução desses casos maiores foram medidos utilizando System.nanoTime(), demonstrando que o algoritmo mantém sua eficiência mesmo em contextos mais exigentes. Esses testes comprovam que a solução é eficiente e capaz de lidar com diferentes variações do problema de maneira satisfatória.

3.1 Capturas de telas contendo os testes

```
// Testes de funcionamento
Run|Debug
public static void main(String[] args) {
    DetectorSubsequencia detector = new DetectorSubsequencia();

    /**
     * Caso 1: Subsequência válida
     * Deve retornar: true
     */
    String[] S1 = {"buy Amazon", "buy Google", "buy Apple", "buy Google", "buy Google", "buy NVIDIA"};
    String[] S_line1 = {"buy Google", "buy Apple", "buy Google", "buy NVIDIA"};
    System.out.println("Caso 1: " + detector.hasTrend(S1, S_line1));

    /**
     * Caso 2: S_line não é subsequência por ser uma ordem incorreta
     * Deve retornar: false
     */
    String[] S2 = {"buy Amazon", "buy Google", "buy Apple", "buy Google", "buy Google", "buy NVIDIA"};
    String[] S_line2 = {"buy Google", "buy NVIDIA", "buy Google"};
    System.out.println("Caso 2: " + detector.hasTrend(S2, S_line2));

    /**
     * Caso 3: S_line é subsequência completa de S
     * Deve retornar: true
     */
    String[] S3 = {"buy Google", "buy Apple", "buy Google", "buy NVIDIA"};
    String[] S_line3 = {"buy Google", "buy Apple", "buy Google", "buy NVIDIA"};
    System.out.println("Caso 3: " + detector.hasTrend(S3, S_line3));
}
```

```

/**
 * Caso 4: S_line é subsequência parcial do começo
 * Deve retornar: true
 */
String[] S_line4 = {"buy Google", "buy Apple"};
System.out.println("Caso 4: " + detector.hasTrend(S1, S_line4));

/**
 * Caso 5: S_line é subsequência parcial do fim
 * Deve retornar: true
 */
String[] S_line5 = {"buy Google", "buy NVIDIA"};
System.out.println("Caso 5: " + detector.hasTrend(S1, S_line5));

/**
 * Caso 6: S_line não existe em S
 * Deve retornar: false
 */
String[] S_line6 = {"buy Tesla"};
System.out.println("Caso 6: " + detector.hasTrend(S1, S_line6));

/**
 * Caso 7: S_line vazia (sempre subsequência)
 * Deve retornar: true
 */
String[] S_line7 = {};
System.out.println("Caso 7: " + detector.hasTrend(S1, S_line7));

```

```

/**
 * Caso 8: S vazia e S_line não vazia
 * Deve retornar: false
 */
String[] S8 = {};
String[] S_line8 = {"buy Google"};
System.out.println("Caso 8: " + detector.hasTrend(S8, S_line8));

/**
 * Caso 9: Ambos vazios, então é uma subsequência válida
 * Deve retornar: true
 */
String[] S9 = {};
String[] S_line9 = {};
System.out.println("Caso 9: " + detector.hasTrend(S9, S_line9));

/**
 * Caso 10: Eventos repetidos em S, mas S_line exige ordem específica
 * Deve retornar: true
 */
String[] S10 = {"buy Google", "buy Google", "buy Google"};
String[] S_line10 = {"buy Google", "buy Google"};
System.out.println("Caso 10: " + detector.hasTrend(S10, S_line10));

/**
 * Caso 11: Ordem errada em S_line
 * Deve retornar: false
 */
String[] S11 = {"buy Amazon", "buy Google", "buy Apple"};
String[] S_line11 = {"buy Apple", "buy Google"};
System.out.println("Caso 11: " + detector.hasTrend(S11, S_line11));

```

```

/**
 * Caso 12: Sequência grande
 * 100 tipos de ações, repetindo
 * Deve retornar: true
 */
String[] S12 = new String[1000000];
for (int i = 0; i < S12.length; i++) {
    S12[i] = "buy Teste" + (i % 100);
}

String[] S_line12 = {"buy Teste1", "buy Teste2", "buy Teste3"};

long tempoInicial12 = System.nanoTime();
System.out.println("Caso 12: " + detector.hasTrend(S12, S_line12));
long tempoFinal12 = System.nanoTime();
System.out.println("Tempo de execução (nanossegundos) Caso 12: " + (tempoFinal12 - tempoInicial12));

/**
 * Caso 13: Pior caso, onde não existe
 * Deve retornar: false
 */
String[] S_line13 = {"buy NaoExiste1", "buy NaoExiste2"};

long tempoInicial13 = System.nanoTime();
System.out.println("Caso 13 (pior caso): " + detector.hasTrend(S12, S_line13));
long tempoFinal13 = System.nanoTime();
System.out.println("Tempo de execução (nanossegundos) Caso 13: " + (tempoFinal13 - tempoInicial13));
}

```

3.2. Casos 12 e 13

Os casos 12 e 13 foram projetados para testar o algoritmo em situações de alta carga de dados, avaliando seu desempenho em termos de tempo de execução e eficiência prática. No Caso 12, foi criada uma sequência S com 1.000.000 de elementos, simulando uma situação realista de séries temporais extensas. Nessa sequência, foram inseridos repetidamente 100 tipos de eventos ("buy Teste0" até "buy Teste99"), de modo que os eventos de interesse da subsequência S_line "buy Teste1", "buy Teste2" e "buy Teste3" certamente ocorrem em ordem dentro de S. Esse teste valida se o algoritmo consegue identificar corretamente uma subsequência em uma grande massa de dados e mede o tempo necessário para isso. Já o Caso 13 simula o pior cenário, onde a subsequência procurada não existe em S. Nesse caso, a subsequência S_line contém eventos fictícios ("buy NaoExiste1", "buy NaoExiste2") que não aparecem em S. Aqui, o algoritmo precisa percorrer toda a sequência de 1.000.000 de elementos sem encontrar os eventos, o que testa seu comportamento de pior caso em tempo linear $O(n)$. Ambos os casos ajudam a comprovar a eficiência do algoritmo tanto em cenários típicos quanto em situações limite, demonstrando que ele é adequado para aplicações com grandes volumes de dados.

4. Tempo de Execução do Algoritmo 1

Os testes de desempenho com grandes volumes de dados demonstraram a eficiência do algoritmo proposto. No Caso 12, onde a subsequência foi encontrada dentro de uma sequência de 1.000.000 de elementos, o tempo de execução variou entre 0,051 ms e 0,101 ms (milissegundos), evidenciando que o algoritmo consegue identificar rapidamente padrões mesmo em grandes entradas. No Caso 13, representando o pior

cenário, em que a subsequência não existe e todo o vetor precisa ser percorrido, o tempo de execução foi naturalmente maior, variando entre 11,445 ms e 12,528 ms. Esses resultados confirmam o comportamento linear do algoritmo e sua eficiência prática em aplicações de séries temporais extensas.

```
[Running] cd "/home/caroline/puc/impl_alg/" && javac DetectorSubsequencia.java && java DetectorSubsequencia
Caso 1: true
Caso 2: false
Caso 3: true
Caso 4: true
Caso 5: true
Caso 6: false
Caso 7: true
Caso 8: false
Caso 9: true
Caso 10: true
Caso 11: false
Caso 12: true
Tempo de execução (nanossegundos) Caso 12: 84599
Caso 13 (pior caso): false
Tempo de execução (nanossegundos) Caso 13: 12528264

[Done] exited with code=0 in 0.657 seconds
```

```
[Running] cd "/home/caroline/puc/impl_alg/" && javac DetectorSubsequencia.java && java DetectorSubsequencia
Caso 1: true
Caso 2: false
Caso 3: true
Caso 4: true
Caso 5: true
Caso 6: false
Caso 7: true
Caso 8: false
Caso 9: true
Caso 10: true
Caso 11: false
Caso 12: true
Tempo de execução (nanossegundos) Caso 12: 101012
Caso 13 (pior caso): false
Tempo de execução (nanossegundos) Caso 13: 11445504

[Done] exited with code=0 in 0.607 seconds
```

5. Problema 2

O segundo problema proposto trata da multiplicação de matrizes, uma operação fundamental em diversas áreas da computação. Até 1969, acreditava-se que o tempo mínimo necessário para multiplicar duas matrizes quadradas de ordem n era $\Theta(n^3)$, utilizando o algoritmo clássico. No entanto, Volker Strassen demonstrou que é possível reduzir essa complexidade para aproximadamente $O(n^{2.81})$, utilizando uma abordagem de divisão e conquista. Esse avanço permitiu realizar multiplicações de matrizes de forma mais eficiente, especialmente em contextos onde o custo computacional é crítico. O problema apresentado na disciplina de Projeto e Otimização de Algoritmos visa explorar essa melhoria, analisando tanto o seu funcionamento quanto a sua eficiência em comparação com métodos tradicionais.

6. Desenvolvimento da Solução e Implementação

A solução foi desenvolvida com base no algoritmo de Strassen, que divide as matrizes A e B em submatrizes menores, realizando operações de soma, subtração e multiplicações específicas para reduzir o número de multiplicações diretas necessárias.

A classe MultiplicacaoStrassen contém o método principal multiply, que recebe duas matrizes quadradas como parâmetros e chama o método recursivo strassen. A função strassen verifica se o tamanho da matriz é 1x1, caso em que realiza a multiplicação diretamente (caso base da recursão).

```
1 public class MultiplicacaoStrassen {
2
3     public int[][] multiply(int[][] A, int[][] B) {
4         int n = A.length;
5         return strassen(A, B, n);
6     }
7
8     private int[][] strassen(int[][] A, int[][] B, int n) {
9         int[][] resultado = new int[n][n];
10
11         // Caso base: matriz 1x1
12         if (n == 1) {
13             resultado[0][0] = A[0][0] * B[0][0];
14         } else {
```

Caso contrário, a matriz é dividida em quatro quadrantes (A11, A12, A21, A22 para a matriz A, e B11, B12, B21, B22 para a matriz B) utilizando a função auxiliar dividirMatriz.

```
        } else {
            int novoTamanho = n / 2;

            int[][] A11 = new int[novoTamanho][novoTamanho];
            int[][] A12 = new int[novoTamanho][novoTamanho];
            int[][] A21 = new int[novoTamanho][novoTamanho];
            int[][] A22 = new int[novoTamanho][novoTamanho];

            int[][] B11 = new int[novoTamanho][novoTamanho];
            int[][] B12 = new int[novoTamanho][novoTamanho];
            int[][] B21 = new int[novoTamanho][novoTamanho];
            int[][] B22 = new int[novoTamanho][novoTamanho];

            // Dividindo as matrizes em 4 submatrizes
            dividirMatriz(A, A11, linha:0, coluna:0);
            dividirMatriz(A, A12, linha:0, novoTamanho);
            dividirMatriz(A, A21, novoTamanho, coluna:0);
            dividirMatriz(A, A22, novoTamanho, novoTamanho);

            dividirMatriz(B, B11, linha:0, coluna:0);
            dividirMatriz(B, B12, linha:0, novoTamanho);
            dividirMatriz(B, B21, novoTamanho, coluna:0);
            dividirMatriz(B, B22, novoTamanho, novoTamanho);
```

Em seguida, são calculadas sete multiplicações intermediárias (M1 a M7) conforme o algoritmo de Strassen, utilizando chamadas recursivas ao próprio método strassen, combinando somas e subtrações das submatrizes com as funções somar e subtrair. Os resultados dessas multiplicações são utilizados para compor os quadrantes finais (C11, C12, C21, C22) da matriz resultado, que são unidos na matriz final através da função juntarMatriz.

```
dividirMatriz(B, B22, novoTamanho, novoTamanho);

// Calculando os 7 produtos de Strassen
int[][] M1 = strassen(somar(A11, A22), somar(B11, B22), novoTamanho);
int[][] M2 = strassen(somar(A21, A22), B11, novoTamanho);
int[][] M3 = strassen(A11, subtrair(B12, B22), novoTamanho);
int[][] M4 = strassen(A22, subtrair(B21, B11), novoTamanho);
int[][] M5 = strassen(somar(A11, A12), B22, novoTamanho);
int[][] M6 = strassen(subtrair(A21, A11), somar(B11, B12), novoTamanho);
int[][] M7 = strassen(subtrair(A12, A22), somar(B21, B22), novoTamanho);

// Calculando os quadrantes do resultado
int[][] C11 = somar(subtrair(somar(M1, M4), M5), M7);
int[][] C12 = somar(M3, M5);
int[][] C21 = somar(M2, M4);
int[][] C22 = somar(subtrair(somar(M1, M3), M2), M6);

// Juntando os quadrantes na matriz resultante
juntarMatriz(C11, resultado, linha:0, coluna:0);
juntarMatriz(C12, resultado, linha:0, novoTamanho);
juntarMatriz(C21, resultado, novoTamanho, coluna:0);
juntarMatriz(C22, resultado, novoTamanho, novoTamanho);
}
```

Para apoiar a implementação do algoritmo de Strassen, foram desenvolvidas quatro funções auxiliares fundamentais para a manipulação de matrizes: dividirMatriz, juntarMatriz, somar e subtrair. A função dividirMatriz é responsável por extrair uma submatriz a partir de uma matriz maior. Ela copia elementos da matriz original origem para a matriz destino, começando de uma posição inicial determinada pelas variáveis linha e coluna, facilitando a divisão das matrizes A e B nos quadrantes A11, A12, A21, A22 e B11, B12, B21, B22. Já a função juntarMatriz realiza o processo inverso: ela insere os elementos de uma matriz menor origem em uma matriz maior destino, também a partir de uma posição inicial específica. Esse método é utilizado para reconstruir a matriz final de resultado após o cálculo dos quadrantes C11, C12, C21 e C22. As funções somar e subtrair são responsáveis por realizar operações de soma e subtração de duas matrizes de mesmo tamanho, elemento a elemento. A função somar percorre todos os índices das matrizes A e B, somando os valores correspondentes e armazenando o resultado em uma nova matriz, enquanto a função subtrair realiza o mesmo procedimento, porém efetuando a subtração entre os elementos.


```
// Funções auxiliares
private void dividirMatriz(int[][] origem, int[][] destino, int linha, int coluna) {
    for (int i = 0; i < destino.length; i++) {
        for (int j = 0; j < destino.length; j++) {
            destino[i][j] = origem[i + linha][j + coluna];
        }
    }
}

private void juntarMatriz(int[][] origem, int[][] destino, int linha, int coluna) {
    for (int i = 0; i < origem.length; i++) {
        for (int j = 0; j < origem.length; j++) {
            destino[i + linha][j + coluna] = origem[i][j];
        }
    }
}
```

```
private int[][] somar(int[][] A, int[][] B) {
    int n = A.length;
    int[][] resultado = new int[n][n];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            resultado[i][j] = A[i][j] + B[i][j];
    return resultado;
}

private int[][] subtrair(int[][] A, int[][] B) {
    int n = A.length;
    int[][] resultado = new int[n][n];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            resultado[i][j] = A[i][j] - B[i][j];
    return resultado;
}
```

A implementação foi projetada para funcionar com matrizes de ordem potência de 2. As capturas de tela apresentadas no relatório ilustram a estrutura do código, a recursividade aplicada, e a manipulação das submatrizes em cada etapa da multiplicação.

6.1 Explicação sobre o desenvolvimento

A estrutura modular do código, com funções específicas para dividir, juntar, somar e subtrair matrizes, facilita a aplicação da estratégia de divisão e conquista, permitindo que o problema original seja progressivamente reduzido até casos simples, solucionados diretamente. Assim, o código desenvolvido atende ao objetivo de otimizar a multiplicação de matrizes respeitando os princípios fundamentais do algoritmo de Strassen.

7. Resultados Obtidos

Os testes realizados com a implementação do algoritmo de Strassen mostraram resultados corretos para todos os casos avaliados, com as matrizes resultantes

conferindo exatamente com os valores esperados. Foram aplicados 10 casos de teste com diferentes combinações de matrizes contendo valores de unidades, dezenas e centenas, assegurando que a multiplicação fosse corretamente realizada em todos os cenários. Em termos de desempenho, os tempos de execução medidos em nanosegundos variaram entre aproximadamente 11.000 ns e 40.000 ns nos dois conjuntos de execuções, o que representa tempos baixos, compatíveis com a baixa ordem das matrizes (2x2). De modo geral, os resultados demonstram que a implementação foi eficiente e fiel ao algoritmo de Strassen, cumprindo tanto a correta multiplicação das matrizes quanto apresentando ótimo desempenho para o tamanho dos dados utilizados.

Resultados:

```
[Running] cd "/home/caroline/puc/impl_alg/" && javac MultiplicacaoSt
Caso 1:
19 22
43 50
Tempo de execução (nanosegundos): 18095

Caso 2:
70 100
150 220
Tempo de execução (nanosegundos): 11887

Caso 3:
1900 2200
4300 5000
Tempo de execução (nanosegundos): 13868

Caso 4:
5 6
7 8
Tempo de execução (nanosegundos): 14554

Caso 5:
130 170
250 330
Tempo de execução (nanosegundos): 11410
```

```
Caso 6:
2900 3300
5700 6500
Tempo de execução (nanosegundos): 32425

Caso 7:
5 5
9 9
Tempo de execução (nanosegundos): 30874

Caso 8:
180 180
420 420
Tempo de execução (nanosegundos): 39861
|
Caso 9:
2745 2388
6297 5496
Tempo de execução (nanosegundos): 35435

Caso 10:
50 67
38 51
Tempo de execução (nanosegundos): 33851

[Done] exited with code=0 in 0.453 seconds
```

Em uma tentativa inicial, foram realizados testes utilizando matrizes com valores muito altos, na casa dos milhares. No entanto, essa abordagem apresentou inconsistências nos resultados, o que pode ser explicado pelo fenômeno de overflow. Como o tipo `int` em Java possui um limite máximo de armazenamento, multiplicações entre números grandes ultrapassam esse valor, resultando em números incorretos. Por isso, para garantir a precisão dos cálculos e evitar erros, optei por ajustar os testes para valores menores, limitando a matrizes com números de unidades, dezenas e centenas.

Para permitir que o algoritmo de Strassen opere corretamente com números grandes sem sofrer overflow, uma solução seria substituir o tipo `int` por `long` em toda a implementação. O tipo `long` em Java suporta valores muito maiores. Para isso, seria necessário alterar todas as assinaturas de métodos, variáveis internas e estruturas de matriz de `int` para `long`. No contexto deste trabalho, apenas trocar para `long` já resolveria o problema de forma eficiente e prática.

7.1. Mudando para long

Mudando tudo de `int` para `long` no código, e mudando os testes para valores maiores, funcionou da forma abaixo:

```

Run | Debug
public static void main(String[] args) {
    MultiplicacaoStrassenLong strassen = new MultiplicacaoStrassenLong();

    long[][][] testesA = {
        {{1000, 2000}, {3000, 4000}},
        {{1500, 2500}, {3500, 4500}},
        {{5000, 6000}, {7000, 8000}},
        {{1234, 5678}, {9101, 1121}},
        {{9999, 8888}, {7777, 6666}},
        {{1111, 2222}, {3333, 4444}},
        {{3210, 6540}, {9870, 1230}},
        {{4000, 5000}, {6000, 7000}},
        {{8000, 7000}, {6000, 5000}},
        {{4321, 1234}, {5678, 4321}}
    };

    long[][][] testesB = {
        {{4000, 3000}, {2000, 1000}},
        {{3500, 2500}, {1500, 500}},
        {{8000, 7000}, {6000, 5000}},
        {{1111, 2222}, {3333, 4444}},
        {{5555, 4444}, {3333, 2222}},
        {{1234, 2345}, {3456, 4567}},
        {{1111, 2222}, {3333, 4444}},
        {{7000, 6000}, {5000, 4000}},
        {{3000, 4000}, {5000, 6000}},
        {{8765, 9876}, {6543, 7654}}
    };
};

```

Resultados:

```
[Running] cd "/home/caroline/puc/impl_alg/" && javac MultiplicacaoStrassenLong.java && java MultiplicacaoStrassenLong
Caso 1:
8000000 5000000
20000000 13000000
Tempo de execução (nanosegundos): 21470

Caso 2:
9000000 5000000
19000000 11000000
Tempo de execução (nanosegundos): 11656

Caso 3:
76000000 65000000
104000000 89000000
Tempo de execução (nanosegundos): 11240

Caso 4:
20295748 27974980
13847504 25204146
Tempo de execução (nanosegundos): 21888

Caso 5:
85168149 64184692
65419013 49372840
Tempo de execução (nanosegundos): 27632
```

```
Caso 6:
9050206 12753169
19471386 28111633
Tempo de execução (nanosegundos): 13930

Caso 7:
25364130 36196380
15065160 27397260
Tempo de execução (nanosegundos): 30387

Caso 8:
53000000 44000000
77000000 64000000
Tempo de execução (nanosegundos): 32779

Caso 9:
59000000 74000000
43000000 54000000
Tempo de execução (nanosegundos): 35221

Caso 10:
45947627 52119232
78039973 89148862
Tempo de execução (nanosegundos): 26525
```

Assim, o problema foi resolvido e continuou funcional sem perder eficiência.

8. Conclusão

O trabalho possibilitou aplicar técnicas de otimização de algoritmos, usando abordagens greedy e divisão e conquista. No primeiro problema, foi desenvolvida uma solução linear eficiente para detectar subsequências. No segundo, a implementação do algoritmo de Strassen permitiu reduzir a complexidade da multiplicação de matrizes. Os testes

validaram a precisão e a eficiência das soluções, reforçando a importância da escolha adequada de algoritmos para cada tipo de problema.