Trabalho 1 - Projeto e Otimização de algoritmos

Leonardo Heinen Oliveira
Leonardo Vargas Soares
Pontifícia Universidade Católica - PUCRS
Professor Rafael Scoppel Silva

22 de abril de 2024

Resumo

Este trabalho tem como objetivo implementar e analisar soluções para os problemas propostos pelo professor. Cada abordagem é analisada separadamente, afim de minuciarmos o processo de análise de cada algoritmo. As seções são divididas de acordo com os problemas (1 - Problema 1, 2 - Problema 2).

1 O Problema (1)

A análise de séries temporais desempenha um papel fundamental na compreensão dos padrões subjacentes aos eventos que se desenrolam ao longo do tempo. Em particular, no campo da mineração de dados financeiros, a detecção de padrões em sequências de eventos, como transações em bolsas de valores, é de grande interesse para investidores, analistas e pesquisadores. Neste contexto, surge o desafio de identificar de forma eficiente certos padrões em sequências extensas de eventos, permitindo a rápida detecção de subsequências relevantes.

Este estudo propõe uma abordagem para a detecção de subsequências relevantes em séries temporais financeiras, visando facilitar a identificação de padrões significativos. Consideramos uma coleção de eventos possíveis e uma sequência específica de eventos que se deseja identificar em uma série temporal maior. O objetivo é determinar se a sequência menor está contida na série maior, mesmo que os eventos não ocorram em ordem consecutiva.

Em suma, a detecção de subsequências relevantes em séries temporais financeiras é crucial para a análise e interpretação de padrões em transações de mercado. Nossa abordagem oferece uma solução eficiente e eficaz para esse desafio, capacitando investidores e analistas a identificar rapidamente padrões significativos e tomar decisões informadas em um ambiente financeiro complexo e em constante evolução.

2 O Algoritmo (1)

2.1 Algoritmos Gulosos - descrição

Os algoritmos gulosos são uma classe importante de algoritmos usados para resolver problemas de otimização em computação. Em sua essência, esses algoritmos fazem escolhas locais que parecem ser as melhores em cada etapa, na esperança de alcançar uma solução globalmente ótima. Eles são chamados de "gulosos"porque, em cada etapa, escolhem a opção que parece ser a mais vantajosa naquele momento, sem considerar consequências futuras.

Uma das características marcantes dos algoritmos gulosos é a falta de retrocesso. Ou seja, uma vez que uma escolha é feita, ela não é reconsiderada posteriormente. Isso pode tornar esses algoritmos simples e eficientes em muitos casos, especialmente quando o problema possui subestrutura ótima e a escolha gulosa leva a uma solução globalmente ótima.

No entanto, é importante ressaltar que nem todos os problemas podem ser resolvidos com algoritmos gulosos, pois nem sempre a escolha localmente ótima leva a uma solução globalmente ótima. Portanto, ao usar algoritmos gulosos, é crucial entender as propriedades do problema em questão e garantir que a estratégia gulosa leve a uma solução correta e eficiente. Para consultar exemplos, explicações mais detalhadas, fórmulas e aplicações de algoritmos gulosos consulte [Roc04].

2.2 Heurística escolhida

A heurística da escolha ótima local neste contexto se baseia na abordagem de buscar cada elemento da subsequência reduzida na sequência original, minimizando o número de comparações necessárias. Essa heurística se beneficia da observação de que, se não conseguirmos encontrar um elemento da subsequência em uma determinada posição na sequência original, não precisamos continuar procurando a partir dessa posição, pois a subsequência deve ser encontrada em ordem.

Essa heurística está relacionada à solução global do problema, pois ao buscar a subsequência reduzida de maneira otimizada na sequência original, podemos determinar rapidamente se a subsequência é uma parte dessa sequência. Essa abordagem não apenas simplifica o processo de verificação, mas também contribui para a eficiência computacional, uma vez que minimiza o número de comparações necessárias para determinar se a subsequência é uma parte da sequência original. Dessa forma, a heurística da escolha ótima local contribui para a obtenção de uma solução precisa e eficiente para o problema global de verificação da subsequência.

2.3 Funcionamento

O algoritmo implementado busca determinar se uma subsequência específica de eventos está contida em uma série temporal maior. Utilizando uma abordagem gulosa, o algoritmo faz escolhas locais que visam minimizar o número de comparações necessárias para identificar a subsequência na série temporal. Em cada passo do algoritmo, percorremos a subsequência reduzida e tentamos encontrar cada elemento na série temporal maior. Ao fazer isso, mantemos um índice para acompanhar a posição atual na série temporal. Se um elemento da subsequência é encontrado na série temporal, avançamos para o próximo elemento na subsequência e continuamos a busca a partir da próxima posição na série temporal. Essa abordagem permite que o algoritmo detecte rapidamente se a subsequência está presente na série temporal, mesmo que os eventos não estejam em ordem consecutiva. Ao evitar retrocessos e otimizar a busca pela subsequência, o algoritmo proporciona uma solução eficiente para identificação de padrões em séries temporais, contribuindo para uma análise informada e decisões eficazes em mercados financeiros dinâmicos.

O algoritmo implementado tem a seguinte assinatura:

```
1 public static boolean hasTrend(String[] S, String[] S_line)
```

3 Análise do algoritmo (1)

3.1 Estruturas de Dados Utilizadas

O algoritmo utiliza duas estruturas de dados principais: ArrayLists para armazenar as sequências de eventos. Essas estruturas dinâmicas são escolhas adequadas para a manipulação de sequências de tamanho variável, permitindo inserções e remoções eficientes.

3.2 Análise Assintótica

A análise do tempo de execução do algoritmo é crucial para entender sua eficiência em diferentes conjuntos de dados. Vamos considerar os principais passos do algoritmo:

1. Percorrer a subsequência reduzida: Este passo possui complexidade $O(m)$, onde m é o tamanho da subsequência reduzida.

2. Para cada elemento da subsequência, percorrer a série temporal maior: No pior caso, este passo tem complexidade $O(n * m)$, onde n é o tamanho da série temporal maior e m é o tamanho da subsequência reduzida.

3. Como o algoritmo pode parar assim que uma subsequência é encontrada, o pior caso ocorre quando a subsequência não está presente na série temporal maior, resultando em uma complexidade total de $O(n * m)$.

Portanto, a complexidade assintótica do algoritmo é $O(n*m)$, onde n é o tamanho da série temporal maior e m é o tamanho da subsequência reduzida. Isso significa que o tempo de execução do algoritmo aumenta linearmente com o tamanho das sequências, o que é uma eficiência aceitável para muitas aplicações práticas. No entanto, para grandes conjuntos de dados, é importante considerar técnicas de otimização ou algoritmos alternativos para garantir um desempenho satisfatório.

4 Implementação e tempo de execução (1)

4.1 Implementação detalhada

- **Inicialização das Estruturas de Dados:**
  - O algoritmo começa convertendo as sequências de eventos, representadas como arrays de strings, em ArrayLists para facilitar a manipulação.
- **Percorrendo a Subsequência Reduzida:**
  - O algoritmo itera sobre cada elemento da subsequência reduzida. Isso é feito usando um loop for, onde a variável de iteração é incrementada de 0 até o comprimento da subsequência reduzida menos um.
- **Busca na Série Temporal Maior:**
  - Para cada elemento da subsequência reduzida, o algoritmo tenta encontrar uma correspondência na série temporal maior. Isso é feito percorrendo a série temporal a partir de uma posição inicial, que é atualizada conforme novas correspondências são encontradas.
  - O algoritmo utiliza outro loop for para percorrer a série temporal a partir da posição atual. Dentro deste loop, compara-se cada elemento da série temporal com o elemento da subsequência reduzida atualmente sendo verificado.
- **Atualização da Posição na Série Temporal:**
  - Após encontrar uma correspondência para um elemento da subsequência reduzida na série temporal, o algoritmo atualiza a posição inicial na série temporal para a próxima posição após a correspondência encontrada. Isso permite que o algoritmo continue a busca a partir dessa posição na próxima iteração do loop da subsequência reduzida.
- **Conclusão da Busca:**
  - Se todos os elementos da subsequência reduzida são encontrados na série temporal maior, o algoritmo retorna verdadeiro, indicando que a subsequência está presente na série temporal. Caso contrário, retorna falso.
- **Complexidade Temporal:**
  - A complexidade temporal do algoritmo é determinada principalmente pelo número de elementos nas sequências: o tamanho da subsequência reduzida (m) e o tamanho da série temporal maior (n). Portanto, a complexidade assintótica do algoritmo é $O(n * m)$.

O passo-a-passo, resulta no seguinte algoritmo:

```java
public static boolean hasTrend(String[] S, String[] S_line) {
    // Medindo o tempo do ínicio do algoritmo
    long start = System.currentTimeMillis();
    ArrayList<String> newS = new ArrayList<>();
    ArrayList<String> newS_line = new ArrayList<>();
    Collections.addAll(newS, S);
    Collections.addAll(newS_line, S_line);      You, 1 second ago • Uncommitted changes
    int indexS = 0; // Índice para percorrer a lista original S

    // Percorre a lista de subsequência reduzida
    for (String s : newS_line) {
        boolean found = false; // Flag para indicar se a string da subsequência foi encontrada

        // Percorre a lista original S a partir do índice atual
        for (int i = indexS; i < newS.size(); i++) {
            if (newS.get(i).equals(s)) { // Se encontrar a string na lista original
                found = true;
                indexS = i + 1; // Atualiza o índice para começar a próxima busca após esta posição
                break;
            }
        }

        // Se a string da subsequência não foi encontrada na lista original, retorna falso
        if (!found) {
            long elapsed = System.currentTimeMillis() - start;
            System.out.printf(format:"Problema 1 - Executado em: %.3f", elapsed);
            return false;

        }
    }

    // Se percorreu toda a lista de subsequência reduzida e encontrou todas as strings na lista original
    // então retorna verdadeiro
    long elapsed = System.currentTimeMillis() - start;
    System.out.printf(format:"Problema 1 - Executado em: %.3f", elapsed);
    return true;
}
```

Figura 1: Algoritmo implementado, na linguagem Java

4.2 Tempo de execução

Utilizamos três conjuntos de dados (pequeno, médio e grande) com diferentes quantidades de elementos(8, 100 e 100) e executamos o algoritmo de detecção de subsequências em cada conjunto três vezes. Em seguida, analisamos o tempo de execução médio em relação ao tamanho dos dados de entrada para visualizar o comportamento assintótico do algoritmo. Utilizando a média das três execuções, geramos o seguinte gráfico:

Figura 2: Gráfico - tempo de execução

5 O Problema (2)

A multiplicação de matrizes é uma operação fundamental em álgebra linear e é comumente utilizada em diversas áreas da matemática, ciência da computação e engenharia. O algoritmo tradicional de multiplicação de matrizes é baseado em uma abordagem simples de três loops aninhados, resultando em uma complexidade assintótica de $O(n^3)$, onde n é a ordem das matrizes. No entanto, a multiplicação de matrizes de V. **Strassen** propõe uma abordagem alternativa que utiliza a técnica de divisão e conquista para reduzir a complexidade assintótica para $O(n^2.81)$.

A principal diferença entre o algoritmo comum e o algoritmo de **Strassen** reside na estratégia de divisão e conquista empregada pelo último. Enquanto o algoritmo tradicional divide as matrizes em pequenos subconjuntos e as multiplica diretamente, o algoritmo de **Strassen** divide as matrizes em submatrizes menores e realiza multiplicações parciais utilizando combinações de adições e subtrações. Isso reduz o número total de multiplicações necessárias, contribuindo para uma menor complexidade assintótica.

Outra diferença crucial é o ponto de corte para a aplicação do algoritmo de **Strassen**. Enquanto o algoritmo tradicional é eficiente para matrizes de tamanho relativamente pequeno, o algoritmo de **Strassen** mostra seu desempenho superior para matrizes maiores, onde a complexidade assintótica se torna mais significativa. Isso ocorre porque a técnica de divisão e conquista de **Strassen** é mais eficaz quando aplicada a matrizes grandes, onde o número de chamadas recursivas (de 8 para 7) pode ser significativamente reduzido.

Embora a multiplicação de matrizes de **Strassen** ofereça vantagens em termos de complexidade assintótica para matrizes grandes, é importante notar que a implementação prática pode não ser sempre mais rápida devido a fatores como a sobrecarga devido às operações adicionais de adição e subtração e as constantes envolvidas no algoritmo. Assim, a escolha entre o algoritmo comum e o algoritmo de **Strassen** depende do tamanho das matrizes envolvidas e das características específicas do sistema em questão. Para exemplificações do funcionamento do algoritmo, consulte [Hok14].

6 O Algoritmo (2)

6.1 Divisão e Conquista

A estratégia de divisão e conquista é um paradigma fundamental em ciência da computação, usado para resolver problemas dividindo-os em subproblemas menores e mais simples. Esses subproblemas são então resolvidos de forma independente, e suas soluções são combinadas para formar a solução do problema original. A ideia básica por trás da divisão e conquista é que, se um problema é suficientemente pequeno, podemos resolvê-lo diretamente sem dividir ainda mais. Se for maior, podemos dividi-lo em subproblemas menores e resolver recursivamente cada um deles. Uma vez que os subproblemas são resolvidos, suas soluções são combinadas para formar a solução do problema maior.

Um exemplo é o *merge sort* é um algoritmo de ordenação eficiente que segue o paradigma de divisão e conquista. Sua operação é baseada em dividir a lista não ordenada em metades, ordenar cada metade de forma independente e, em seguida, combinar as metades ordenadas para obter a lista final ordenada.

Outro exemplo comum é o algoritmo de busca binária, que é usado para encontrar um elemento em uma lista ordenada. Nesse caso, o problema de busca é dividido em metades a cada passo, eliminando metade dos elementos restantes em cada iteração. Isso resulta em um tempo de execução muito eficiente, já que o número de elementos a serem considerados é reduzido pela metade a cada passo. Em geral, a estratégia de divisão e conquista é amplamente utilizada na ciência da computação devido à sua eficácia e versatilidade na resolução de uma variedade de problemas complexos.

6.2 Escolha do caso base

No algoritmo de **Strassen** para multiplicação de matrizes, a escolha do caso base é fundamental para aplicar a estratégia de divisão e conquista de maneira eficiente. O caso base é o ponto onde o problema é pequeno o suficiente para ser resolvido diretamente, sem a necessidade de dividir ainda mais em subproblemas.

No caso do algoritmo de **Strassen**, o caso base é frequentemente definido como matrizes de dimensões pequenas, geralmente 1x1 ou 2x2. Isso ocorre porque a multiplicação direta de matrizes pequenas é mais eficiente do que continuar dividindo em submatrizes menores.

```
// Caso base para a multiplicação (tamanho da matriz 1x1)
if (tamanho == 1) {
    int[][] C = new int[1][1];
    C[0][0] = A[0][0] * B[0][0];
    return C;
}
```

Figura 3: Caso base do algoritmo implementado

6.3 Funcionamento

Ao atingir o caso base, o algoritmo de **Strassen** realiza a multiplicação direta das matrizes de dimensões pequenas, evitando assim a recursão adicional. Isso reduz o custo de recursão e a sobrecarga associada à divisão e conquista para problemas de tamanho pequeno.

Relacionando isso com a estratégia de divisão e conquista, podemos ver que o algoritmo de **Strassen** aplica a divisão e conquista dividindo matrizes de dimensões maiores em submatrizes menores até atingir o caso base. Em seguida, ele utiliza a etapa de conquista para combinar as soluções das submatrizes menores, obtendo assim a solução final.

Portanto, ao escolher um caso base adequado, o algoritmo de **Strassen** pode aproveitar ao máximo a

estratégia de divisão e conquista, reduzindo eficientemente o custo computacional da multiplicação de matrizes para matrizes grandes. Isso demonstra como a escolha cuidadosa do caso base é crucial para o sucesso da aplicação da estratégia de divisão e conquista em problemas algorítmicos.

O algoritmo implementado tem a seguinte assinatura:

```
1 public static int[][] multiply(int[][] A, int[][] B)
```

7 Análise do algoritmo (2)

7.1 Análise do Algoritmo de Strassen

- O algoritmo de **Strassen** utiliza a estratégia de divisão e conquista para reduzir o número de operações de multiplicação de matrizes, substituindo algumas dessas operações por operações de soma e subtração.
- A complexidade assintótica do algoritmo de **Strassen** é expressa pela equação de $T(n) = 7T\left(\frac{n}{2}\right) + O(n^2)$, onde n é a dimensão das matrizes de entrada e $T(n)$ representa o tempo necessário para multiplicar duas matrizes de tamanho $n \times n$.
- De acordo com o teorema mestre, a complexidade assintótica do algoritmo de **Strassen** é $O(n^{\log_2 7}) \approx O(n^{2.81})$.

7.2 Análise do Algoritmo Comum de Multiplicação de Matrizes

- O algoritmo comum de multiplicação de matrizes segue uma abordagem direta, onde cada elemento da matriz resultante é obtido pela soma dos produtos dos elementos correspondentes das linhas da primeira matriz e das colunas da segunda matriz.
- A complexidade assintótica do algoritmo comum de multiplicação de matrizes é $O(n^3)$, onde n é a dimensão das matrizes de entrada.

7.3 Comparação

- A análise assintótica do algoritmo de **Strassen** mostra que ele possui uma complexidade menor do que a multiplicação de matrizes convencional para dimensões $n > 2$. Isso se deve à redução no número de operações de multiplicação direta.
- No entanto, o algoritmo de **Strassen** tem uma sobrecarga adicional devido às operações de soma e subtração, bem como a combinação dos resultados intermediários, o que resulta em uma complexidade assintótica maior do que a multiplicação de matrizes convencional para matrizes de tamanho pequeno ou moderado.
- Portanto, enquanto o algoritmo de **Strassen** oferece ganhos significativos de eficiência para matrizes grandes, o algoritmo comum de multiplicação de matrizes pode ser mais eficiente para matrizes de tamanho pequeno a moderado.

8 Implementação e Tempo de Execução

8.1 Divisão das Matrizes

O algoritmo começa dividindo as matrizes de entrada em quatro submatrizes de igual tamanho. Para matrizes quadradas de tamanho $n \times n$, cada matriz é dividida em quatro submatrizes $\frac{n}{2} \times \frac{n}{2}$, representadas como A_{11}, A_{12}, A_{21}, A_{22}, B_{11}, B_{12}, B_{21}, B_{22}.

8.2 Recursão:

Após a divisão das matrizes, o algoritmo recursivamente multiplica essas submatrizes. Isso é feito dividindo repetidamente as matrizes até que alcancemos um caso base, geralmente matrizes de tamanho 1x1 ou 2x2. Para matrizes de tamanho maior, o algoritmo continua dividindo as matrizes em submatrizes menores até atingir o caso base.

8.3 Multiplicação das Submatrizes:

Em cada chamada recursiva, o algoritmo de Strassen multiplica as submatrizes de acordo com as fórmulas específicas. Essas fórmulas envolvem sete produtos intermediários (P1 a P7), que são calculados usando operações de soma e subtração.

8.4 Combinação dos Resultados:

Após calcular os produtos intermediários, o algoritmo combina esses resultados para obter as submatrizes da matriz resultante. As submatrizes resultantes são então combinadas para formar a matriz final do produto.

8.5 Retorno do Resultado:

Por fim, o algoritmo retorna a matriz resultante, que é o produto das matrizes de entrada.

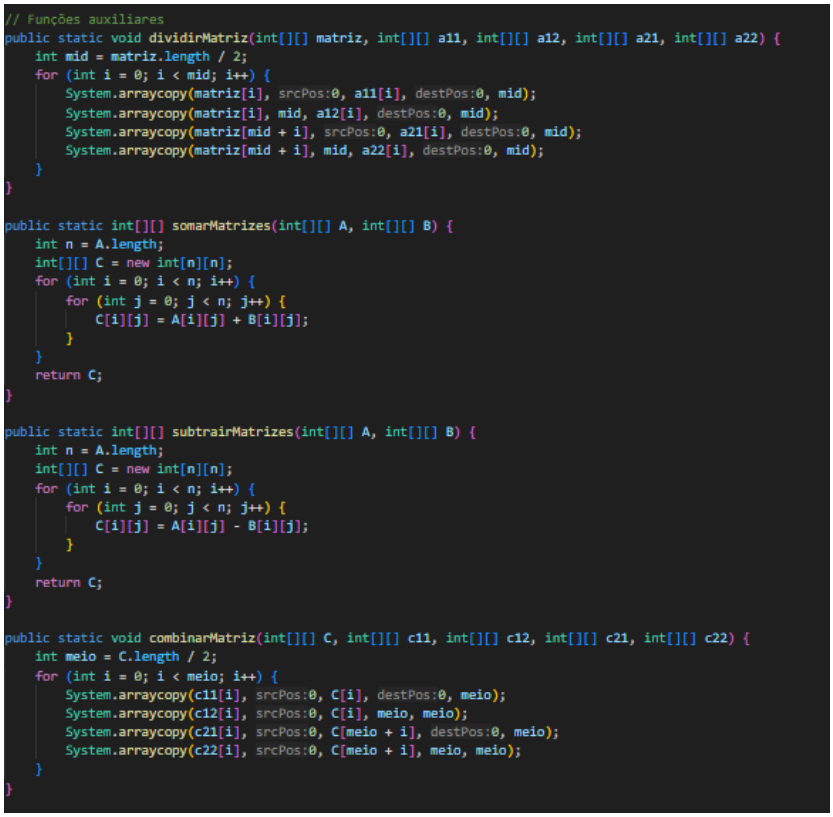
No caso específico das matrizes de exemplo $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ e $B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$, o algoritmo de **Strassen** realizará os seguintes passos:

- Divide as matrizes A e B em quatro submatrizes cada.
- Multiplica as submatrizes recursivamente.
- Combina os resultados intermediários para obter a matriz final do produto.
- Retorna a matriz resultante, que é $\begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$.

Resultando no seguinte algoritmo:

```java
public static int[][] multiply(int[][] A, int[][] B){
    int tamanho = A.length;

    // Caso base para a multiplicação (tamanho da matriz 1x1)
    if (tamanho == 1) {
        int[][] C = new int[1][1];
        C[0][0] = A[0][0] * B[0][0];
        return C;
    }

    // Dividir as matrizes em submatrizes
    int[][] A11 = new int[tamanho/2][tamanho/2];
    int[][] A12 = new int[tamanho/2][tamanho/2];
    int[][] A21 = new int[tamanho/2][tamanho/2];
    int[][] A22 = new int[tamanho/2][tamanho/2];
    int[][] B11 = new int[tamanho/2][tamanho/2];
    int[][] B12 = new int[tamanho/2][tamanho/2];
    int[][] B21 = new int[tamanho/2][tamanho/2];
    int[][] B22 = new int[tamanho/2][tamanho/2];

    dividirMatriz(A, A11, A12, A21, A22);
    dividirMatriz(B, B11, B12, B21, B22);

    // Calcular os produtos intermediários P1-P7
    int[][] P1 = multiply(somarMatrizes(A11, A22), somarMatrizes(B11, B22));
    int[][] P2 = multiply(somarMatrizes(A21, A22), B11);
    int[][] P3 = multiply(A11, subtrairMatrizes(B12, B22));
    int[][] P4 = multiply(A22, subtrairMatrizes(B21, B11));
    int[][] P5 = multiply(somarMatrizes(A11, A12), B22);
    int[][] P6 = multiply(subtrairMatrizes(A21, A11), somarMatrizes(B11, B12));
    int[][] P7 = multiply(subtrairMatrizes(A12, A22), somarMatrizes(B21, B22));

    // Calcular os elementos da matriz de resultado
    int[][] C11 = somarMatrizes(subtrairMatrizes(somarMatrizes(P1, P4), P5), P7);
    int[][] C12 = somarMatrizes(P3, P5);
    int[][] C21 = somarMatrizes(P2, P4);
    int[][] C22 = somarMatrizes(subtrairMatrizes(somarMatrizes(P1, P3), P2), P6);

    // Combinar os resultados em uma única matriz
    int[][] C = new int[tamanho][tamanho];
    combinarMatriz(C, C11, C12, C21, C22);


    return C;
}
```

Figura 4: Código do algoritmo de **Strassen**



```java
// Funções auxiliares
public static void dividirMatriz(int[][] matriz, int[][] a11, int[][] a12, int[][] a21, int[][] a22) {
    int mid = matriz.length / 2;
    for (int i = 0; i < mid; i++) {
        System.arraycopy(matriz[i], srcPos:0, a11[i], destPos:0, mid);
        System.arraycopy(matriz[i], mid, a12[i], destPos:0, mid);
        System.arraycopy(matriz[mid + i], srcPos:0, a21[i], destPos:0, mid);
        System.arraycopy(matriz[mid + i], mid, a22[i], destPos:0, mid);
    }
}

public static int[][] somarMatrizes(int[][] A, int[][] B) {
    int n = A.length;
    int[][] C = new int[n][n];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            C[i][j] = A[i][j] + B[i][j];
        }
    }
    return C;
}

public static int[][] subtrairMatrizes(int[][] A, int[][] B) {
    int n = A.length;
    int[][] C = new int[n][n];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            C[i][j] = A[i][j] - B[i][j];
        }
    }
    return C;
}

public static void combinarMatriz(int[][] C, int[][] c11, int[][] c12, int[][] c21, int[][] c22) {
    int meio = C.length / 2;
    for (int i = 0; i < meio; i++) {
        System.arraycopy(c11[i], srcPos:0, C[i], destPos:0, meio);
        System.arraycopy(c12[i], srcPos:0, C[i], meio, meio);
        System.arraycopy(c21[i], srcPos:0, C[meio + i], destPos:0, meio);
        System.arraycopy(c22[i], srcPos:0, C[meio + i], meio, meio);
    }
}
```

Figura 5: Funções auxiliares para o algoritmo

8.6 Tempo de execução

Para testar os algoritmos de multiplicação de matrizes, foram gerados três conjuntos de dados com diferentes tamanhos de entrada. Cada conjunto consiste em duas matrizes preenchidas com números aleatórios:

- Conjunto de Dados Pequeno: Matrizes de tamanho 10×10.
- Conjunto de Dados Médio: Matrizes de tamanho 100×100.
- Conjunto de Dados Grande: Matrizes de tamanho 1000×1000.

Realizando uma média sobre três execuções nestes três conjuntos, temos o seguinte gráfico:

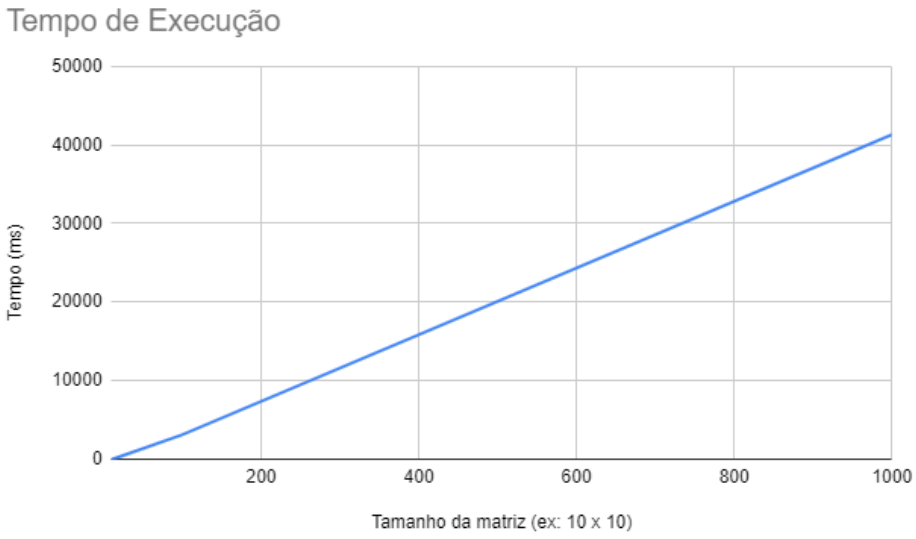


Figura 6: Gráfico de tempo de execução: Tamanho da matriz X Tempo (ms)

Gráfico acima (figura 6) é semelhante ao de uma complexidade O(n), porém algoritmo possui O($n^{2.18}$). Reconhecemos que devido à falta de poder computacional, não conseguimos dar andamento aos testes de tempo de execução deste algoritmo. Com uma massa de dados maior e com tamanhos de matrizes maiores, poderíamos realizar experimentos mais eficazes sobre o tempo de execução do algoritmo.

Referências

[Hok14] Pedro Henrique Del Bianco Hokama. Divisão e conquista - icunicamp. 2014.

[Roc04] Anderson Rocha. Algoritmos gulosos: definições e aplicações. 2004.