1º Trabalho de Projeto e Analise de Algoritmos (PAA)
Métodos de Ordenação
Estratégia Gulosa
Programação Dinâmica

Gabriel Santos da Silva[1], **David Antonio Brocardo**[1]

[1]Centro de Ciências Exatas e Tecnológicas
Campus de Cascavel - UNIOESTE
Caixa Postal 801 – 85.814-110 – Cascavel – PR – Brazil

{gabirel.silva77,david.brocardo}@unioeste.br

Abstract. *This work proposes a detailed empirical investigation with the objective of comparing, from the perspective of asymptotic cost and chronological execution times, two different versions of the MergeSort sorting algorithm, innovative in the C++ programming language. The analysis seeks to evaluate the performance of each approach, considering both theoretical efficiency, based on asymptotic behavior, and practical performance, measured by real execution times in different scenarios and data sets.*

Resumo. *Este trabalho propõe uma investigação empírica detalhada com o objetivo de comparar, sob a ótica do custo assintótico e dos tempos de execução cronológicos, duas versões distintas do algoritmo de ordenação MergeSort, implementadas na linguagem de programação C++. A análise busca avaliar o desempenho de cada abordagem, considerando tanto a eficiência teórica, com base no comportamento assintótico, quanto a performance prática, medida pelos tempos reais de execução em diferentes cenários e conjuntos de dados.*

1. Introdução

A ordenação é uma operação fundamental em computação, amplamente utilizada em uma variedade de aplicações, desde a classificação de listas de números até a organização de dados em bancos de dados [Akhter et al. 2016]. A escolha do algoritmo de ordenação apropriado pode ter um grande impacto no desempenho de um programa, varias analises referentes a métodos de ordenação são amplamente realizadas, como a análise de [Gul et al. 2015] que forneceu uma comparação detalhada de diferentes algoritmos de ordenação em termos de complexidade temporal.

Este trabalho propõe uma investigação empírica para comparar no âmbito de custo assintótico e tempos cronológicos duas versões do método de ordenação MergeSort implementado na linguagem de programação C++. Segundo [Cormen et al. 2009], O MergeSort segue de perto o paradigma de dividir e conquistar. Intuitivamente, funciona da seguinte forma. Dividir: Divida a sequência de n-elementos a ser ordenada em duas subsequências de n/2 elementos cada. Conquistar: ordene as duas subsequências recursivamente usando MergeSort. Combinar: Mesclar as duas subsequências ordenadas para produzir a resposta ordenada. [...] A operação chave do algoritmo MergeSort é a fusão de duas sequências ordenadas na etapa “combine”.

O objetivo deste trabalho é aplicar e corroborar conceitos adquiridos com relação ao método de ordenação MergeSort, bem como elucidar se a passagem por referência, trabalhando com um único vetor em memória e mais vantajosa em relação a criar uma cópia do vetor a cada chamada recursiva do algoritmo.

Para isso foi necessário a implementação de duas variações do MergeSort, primeiro a abordagem clássica do MergeSort que aloca um vetor temporário a cada chamada recursiva e segundo a abordagem alternativa do MergeSort que, ao invés de alocar um novo vetor a cada chamada, simplesmente sobrepõe o vetor de entrada passando por referência. Foi realizados testes empíricos para medir o desempenho e tempos cronológicos das duas versões do MergeSort sobre vários conjuntos de dados diferentes e comparar os resultados dos testes para identificar diferenças significativas de desempenho entre os métodos de ordenação.

Este documento está organizado da seguinte forma: A Seção 2 descreve a implementação dos algoritmos e analisa seu custo assintótico. A Seção 3 detalha o processo de execução e a coleta de dados para os diferentes conjuntos de entrada. Na Seção 4, é apresentada uma análise dos resultados obtidos a partir dos dados coletados. Por fim, a Seção 5 traz as considerações finais do trabalho.

2. Abordagens implementadas

Para atingir o objetivo deste trabalho, de justamente verificar qual abordagem MergeSort é mais adequada para diferentes tipos de dados, seja ela abordagem clássica do MergeSort que aloca um vetor temporário a cada chamada recursiva, ou a abordagem alternativa do MergeSort que, ao invés de alocar um novo vetor a cada chamada, simplesmente sobrepõe o vetor de entrada, também chamada de "*In-Place*". As duas versões foram implementadas utilizando a IDE Visual Studio Code e armazenadas em um repositório na plataforma GitHub. Foi escolhido a linguagem de programação C++ para ambas as implementações.

A linguagem C++ foi escolhida pois ela é uma excelente escolha se tratando de desempenho para aplicações que exigem a máxima velocidade e eficiência, alem de dar um maior controle sobre o hardware e a memória e a flexibilidade em criar um software complexo e adaptável.

Para automatizar o processo de execução, foi desenvolvido um terceiro algoritmo, denominado "ChamaMerge.cpp", que não interfere no desempenho das duas implementações do MergeSort. Sua principal função é compilar e executar os algoritmos do Merge clássico e do Merge alternativo de maneira automatizada. Além disso, o algoritmo ChamaMerge realiza outras tarefas essenciais, como a leitura de um arquivo criado pelos autores, chamado "nomes.txt", que contém o nome do arquivo com os dados a serem processados pelos algoritmos de MergeSort. O ChamaMerge também é responsável por executar cada versão do MergeSort seis vezes sobre o mesmo conjunto de dados. Durante essas execuções, o primeiro tempo de execução é descartado e os tempos das cinco execuções subsequentes são somados, com a média sendo calculada para representar o desempenho do algoritmo. Para a coleta dos tempos de execução, foi utilizada a biblioteca "time", que registra o tempo inicial no momento da execução do algoritmo e o tempo final quando ele é concluído, permitindo assim calcular o tempo total de execução e armazenar esse tempo devidamente identificado, com a versão do MergeSort e o arquivo de entrada em um arquivo csv. O tempo registrado pela biblioteca "time"é em milissegundos.

A parti que do algoritmo “ChamaMerge.cpp”, é executados as duas versões do MergeSort, que serão abordadas nas duas sub seções adiante.

2.1. MergeSort clássico

A implementação base do MergeSort clássico foi pega do repositório do Github do [Brunet 2019], ele que é professor assistente da universidade federal de Campina Grande. Foi decidido usar o código apenas como base, sendo alterado algumas estruturas e variáveis a fim de melhor entendimento dos programadores, a escolha desse repositório foi devido a boa representação da abordagem mais clássica do MergeSort, que é um dos objetivos do corrente trabalho.

Analisando o código, em primeiro lugar, vamos analisar o método MergeSort, “void mergeSort(vector<int>& vetor, int left, int right)”ele é o responsável por dividir o vetor recursivamente na metade até que sobre apenas um elemento. Os parâmetros passados a ele são: o vetor a ser ordenado, um índice *left* e um índice *right* como observamos no trecho a seguir “mergeSort(vetor, 0, n − 1);”. Inicialmente observamos na primeira linha do método, a condição de parada do algoritmo presente no “if (left < right){”, essa condição garante que a porção do algoritmo a ser analisada não possui apenas um elemento, caso o vetor possua somente um elemento, o IF se torna falso e assim se sabe que não há mais a necessidade de continuar a dividi-lo. Em sequência, temos as linhas responsáveis por fazer a divisão e conquista do nosso algoritmo: 1.

Listing 1. Implementação do Merge Sort Clássico - Chamadas recursivas.

```
int mid = left + (right - left) / 2;

//chamadas recursivas para as metades esquerda e direita
mergeSort(vetor, left, mid);
mergeSort(vetor, mid + 1, right);

//mesclando os vetores j  ordenados
merge(vetor, left, mid, right);
```

A primeira define uma variável “mid” como sendo o valor central entre left e right. As próximas duas linhas são chamadas recursivas para o próprio MergeSort, porém a primeira passando como parâmetro o vetor a ser ordenado, o índice left e o índice mid calculado anteriormente; a próxima linha bem semelhante, porém passando como parâmetro o vetor a ser ordenado, o índice mid+1 (essa soma de +1 tem como objetivo evitar o mesmo valor em duas chamadas diferentes) e o índice right. Note que com isso conseguimos quebrar o vetor sempre na metade, até atingir a condição de parada. Por fim, após cada quebra há uma chamada ao método Merge, passando como parâmetro o vetor e os limites a serem considerados: left, mid e right.

Vamos analisar agora o método chamado merge, ele é o responsável por mesclar os vetores já ordenados, inicialmente neste método temos as variáveis n1 e n2 “int n1 = mid − left + 1; int n2 = right − mid;”, elas são responsáveis em calcular o tamanho necessário dos vetores temporários, que são logo declarados como vetor L de tamanho n1 e vetor R de tamanho n2 “vector<int> L(n1), R(n2);”, respectivamente são vetores responsáveis pela parte Left e Right da mesclagem. Em sequência temos o seguinte trecho de código: 2.

Listing 2. Implementação do Merge Sort Clássico - Cópia do conteúdo nos vetores temporários.

```
//copiando para vetor temporario left
for (int i = 0; i < n1; i++)
        L[i] = vetor[left + i];

//copiando para vetor temporario right
for (int j = 0; j < n2; j++)
        R[j] = vetor[mid + 1 + j];
```

Esses dois laços de repetição são responsáveis pela cópia do conteúdo nos vetores temporários, notamos que o intervalo do laço é cuidadosamente selecionado para zero até a variável n1 no laço Left e zero até n2 no laço Right, também é notável que além do índice correto a atribuição deve tomar o cuidado em ser apenas na metade Left para o laço Left e mid+1 para o laço Right.

Em seguida, temos algumas declarações de variáveis que serão usadas na sequência "int i = 0, j = 0, k = left ;"e um laço while responsável por mesclar os vetores temporários de volta ao vetor original, como observamos a seguir: 3.

Listing 3. Implementação do Merge Sort Clássico - Mesclando os vetores temporários.

```
//mesclando os temp vet de volta ao: vetor[left..right]
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            vetor[k] = L[i];
            i++;
        } else {
            vetor[k] = R[j];
            j++;
        }
        k++;

    }
```

O laço while tem como condição que, enquanto não tiver percorrido todas as posições de L ou R, deve continuar mesclando. O IF na sequência compara o elemento atual de L com o elemento atual de R, garantindo que se o elemento de L for menor ou igual ao de R, ele é copiado para a posição correta no vetor original e os índices i e k são incrementados, agora caso o elemento de R for menor do que o de L, o else garante que ele é copiado para o vetor original e o índice j e k é incrementado.

Por fim temos dois while finais: 4.

Listing 4. Implementação do Merge Sort Clássico - Copiando para o vetor original elementos restantes.

```
//se a metade final n o foi toda consumida, faz o append.
    while (i < n1) {
        vetor[k] = L[i];
        i++;
        k++;
    }

//se a metade inicial n o foi toda consumida, faz o append.
```

```
while (j < n2) {
    vetor[k] = R[j];
    j++;
    k++;
}
```

com condição "(i < n1) e (j < n2)"seus funcionamentos são semelhantes e eles tem como objetivo copiar para o vetor original elementos restantes caso um dos subvetores ainda possuir elementos após o while principal, esses elementos são copiados diretamente para o vetor original, já que os elementos já estão ordenados em relação aos elementos que estão copiados.

Com isso, nosso vetor que passamos como parâmetro inicial na chamada do método MergeSort estará ordenado corretamente utilizando o método de ordenação MergeSort clássico.

2.1.1. Polinômio de custo assintótico

Ao realizar uma avaliação do custo assintótico de tempo do algoritmo **MergeSort Clássico**, é necessário analisar o custo linha por linha, considerando as funções "mergesort"e "merge", anteriormente apresentadas.

Logo na função "merge"observamos as linhas "int n1 = mid − left + 1;", partindo do pressuposto que operações como subtração, soma e atribuição tem custo de uma unidade, concluímos que essa linha inicial possui custo constante de 3 (1 operação de subtração + 1 operação de soma + 1 operação de atribuição). A linha "int n2 = right − mid;", possui custo constante de 2 devido a 1 operação de subtração + 1 operação de atribuição. Em sequência "vector<int> L(n1), R(n2);", possui custo linear de n1 + n2 que podemos simplificar para somente **n** devido a inicialização dos vetores dinâmicos de tamanho n1 e n2. Agora possuímos um laço de repetição "for (int i = 0; i < n1; i++) L[i] = vetor[left + i];", o laço itera n1 vezes, realizando uma atribuição por iteração, portanto podemos simplificar seu custo total a **n**. Em seguida possuímos mais um laço de repetição semelhante ao anterior "for (int j = 0; j < n2; j++) R[j] = vetor[mid + 1 + j];", seu custo simplificando é **n**. Agora algumas atribuições com custo constante "int i = 0, j = 0, k = left ;", custo 3. Finalmente nosso laço while principal: 5.

Listing 5. Implementação do Merge Sort Clássico - While principal para merge.

```
while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            vetor[k] = L[i];
            i++;
        } else {
            vetor[k] = R[j];
            j++;
        }
        k++;

    }
```

Devido a 1 comparação para "i $<$ n1"e "j $<$ n2"; 1 comparação para "L[i] $<=$ R[j]"; 1 atribuição; 1 incremento de "i"ou "j"e 1 incremento de "k", o custo total do laço é 5(n1 + n2) que podemos simplificar para **n**. Por fim nossos dois últimos laços, "while (i $<$ n1) { vetor[k] = L[i]; i++; k++; } ", e "while (j $<$ n2) { vetor[k] = R[j]; j++; k++; } ", o primeiro laço possuim custo de 4n devido a 1 comparação para "i $<$ n1"; 1 atribuição; 1 incremento de "i"e 1 incremento de "k", o segundo laço também com custo de 4n, pois ocorre 1 comparação para "j $<$ n2"; 1 atribuição; 1 incremento de "j"e 1 incremento de "k", podemos simplificar ambos para **n**. Com isso somando todos os custos obtemos que a função "merge"tem o polinômio de custo assintótico de total de T(n) = 3 + 2 + n + n + n + 3 + n + n + n, que resulta em O(6n), como O(6n) é linear, podemos simplificar para T(n) = O(n).

Ao examinar a função "mergesort", nota-se que possuímos basicamente custos constantes nas linhas "if (left $<$ right)", custo 1 devido a uma comparação; "if (int mid = left + (right − left) / 2;)", custo 4 devido a uma divisão; uma subtração; uma soma e uma atribuição. E por fim devido à recursão devemos levar em consideração que o problema sempre é reduzido pela metade, então temos duas chamadas recursivas que reduzem o problema pela metade, ou seja, 2T(n/2) mais o custo da linha "merge(vetor, left , mid, right);"que como já analisamos anteriormente tem o custo de O(n). Com isso chegamos em T(n) = 1 + 4 + 2T(n/2) + O(n); que somando todos os custos obtemos que a função "mergesort"tem custo total de T(n) = 5 + 2T(n/2) + O(n).

Portanto o polinômio de custo assintótico, levando em consideração as duas funções principais do algoritmo mergeSort Clássico é: "T(n) = 5 + 2T(n/2) + 3 + 2 + n + n + n + 3 + n + n + n.", que podemos simplificar para uma relação de recorrência de "T(n)=2T(N/2)+N."

Ao realizar uma avaliação do custo assintótico do algoritmo **MergeSort clássico**, a relação de recorrência foi encontrada. Analisando o código e comportamento do algoritmo já aqui apresentado, observamos que o MergeSort clássico possui duas chamadas recursivas, assim reduzindo o problema na metade. Ou seja, "2T(N/2)". Também possuímos uma chamada ao método Merge que sabemos que tem custo O(n). Com todos esses dados em mãos chegamos na relação de recorrência de:

$$T(n) = 2T\left(\frac{N}{2}\right) + N.$$

Agora para o próximo passo basta resolvermos essa relação de recorrência, a fim de praticidade utilizaremos o Teorema Mestre para resolvermos esse problema. Ao aplicar o Teorema Mestre, retiramos o valor 2 da relação, pois possuímos duas chamadas recursivas e apelidamos ele de "a", também retiramos o valor de 2 da relação, pois o problema é reduzido na metade e apelidamos de "b", por fim retiramos o N e apelidamos de "f(n)" já que é o custo presente que temos na relação.

No teorema mestre realizamos uma comparação entre $n^{\log_b a} e f(n)$. Assim podemos cair em três casos diferentes: Caso um, quando o $n^{\log_b a}$ é maior que o f(n). Caso dois, quando eles são iguais e Caso três quando f(n) é maior que $n^{\log_b a}$.

Em cada caso a complexidade se dá assim: Caso um, a complexidade é O($n^{\log_b a}$). Caso dois, a complexidade é O($f(n) * \log_b n$) e Caso três a complexidade é O($f(n)$).

Aplicando o Teorema Mestre em nossa relação de recorrência obtemos $n^{\log_2 2} -> n^1 = n$ e $N = n$. Com isso percebemos que se aplica o caso dois, quando ambos são iguais, então a complexidade se dá pelo O($n * \log_2 n$) que resulta em:

$$O(n \log\ n).$$

Fazendo uma análise mais a fundo percebemos que independente do caso (melhor, pior ou médio), o MergeSort clássico nos garante complexidade O(n log n), pois o vetor sempre será dividido na metade, independente dos dados, estamos sempre dividindo o vetor na metade. Portanto, a complexidade, quando o método MergeSort clássico é usada, sempre nos fornece um custo O(n log n).

2.2. MergeSort alternativo (In-Place)

Analisando a segunda abordagem utilizada, o MergeSort In-Place, baseamo-nos em uma implementação disponibilizada no site [for Geeks 2023], que foi adaptada para atender às necessidades do trabalho. Entre as adequações realizadas, destaca-se a inclusão de um trecho de código responsável pela leitura dos dados do arquivo de entrada e o tratamento necessário para convertê-los ao formato de números inteiros, assim como foi feito no MergeSort padrão. Optou-se pela primeira abordagem apresentada no [for Geeks 2023] por se adequar melhor aos objetivos do trabalho, apesar de possuir o pior custo assintótico entre as opções disponíveis, sendo quadrático, como será discutido posteriormente.

Antes de abordar o cálculo assintótico do algoritmo, é essencial compreender seu funcionamento, analisando-o em termos de ordem de execução. O fluxo inicia-se na função principal (main), que processa os dados de entrada e realiza a chamada do MergeSort In-Place. Essa chamada ocorre através do comando: “ merge_sort_in_place (vetor , 0, vetor . size () − 1);”.

Nessa etapa, o vetor de entrada, delimitado pelos índices inicial e final, é passado por referência para ser manipulado e ordenado.

A função “ merge_sort_in_place ”implementa a estratégia de divisão e conquista característica do MergeSort. O vetor é recursivamente dividido em partes menores até que cada sublista contenha apenas um elemento. Esse processo pode ser observado no trecho de Código 6.

Listing 6. Implementação alternativa do Merge Sort - Função recursiva

```
void merge_sort_in_place(vector<int>& arr, int inicio, int fim) {
    if (inicio < fim) { // 1 comparacao
        int meio = inicio + (fim - inicio) / 2; // 1 atrib + 1 soma + 1 sub + 1 div =4
        merge_sort_in_place(arr, inicio, meio); // 1 chamada rec = 1
        merge_sort_in_place(arr, meio + 1, fim); //1 chamada rec + 1 soma = 2
        merge_in_place(arr, inicio, meio, fim); // 1 chamada de funcao
    }
}
```

A divisão ocorre sempre ao meio, começando pela primeira metade do vetor e prosseguindo até que cada subdivisão tenha um único elemento. Em seguida, a segunda metade do vetor é processada da mesma forma. Após a subdivisão completa, a função “merge_in_place”é chamada para combinar e ordenar as partes. Essa função recebe as informações da subdivisão atual através das variáveis inicio, meio e fim, conforme especificado na chamada, “merge_in_place (arr , inicio , meio, fim);”.

A função “merge_in_place”realiza a fusão das sublistas diretamente no vetor original, evitando o uso de espaço adicional, como acontece na implementação padrão do MergeSort com vetores temporários, como é apresentando no Código 7.

Listing 7. Implementação alternativa do Merge Sort - Função Quadrática

```
void merge_in_place(vector<int>& arr, int inicio, int meio, int fim) {
    int i = inicio;  // 1 atrib = 1
    int j = meio + 1;   // 1 atrib + 1 soma =  2
    while (i <= meio && j <= fim) { // (2 comparacao + 1 teste logico )* n = 3n
        if (arr[i] <= arr[j]) { // (1 comp + 2 indexa  o) * n = 2n
            i++; // n
        } else {
            int aux = arr[j]; // (1 atrib + 1 index) * n =  2n
            for (int k = j; k > i; k--) { // (1 atib*n + ((n^2+n)/2 * (1 comp + 1 dec)) + 1comp *n
= n^2 + 3n
                arr[k] = arr[k - 1]; // (1 atrib + 1 index + 1 sub)* (n^2+n)/2 = 2n^2 + 2n
            }
            arr[i] = aux; // 1 atrib + 1 index = 2n
            i++; // 1 inc = n
            meio++;// 1 inc = n
            j++;// 1 inc = n
        }
    } //Total = 3n^2 + 18n + 3
}
```

O funcionamento dessa função é baseado em dois ponteiros: i, que aponta para o início da primeira sublista, e j, que aponta para o início da segunda sublista. Se o elemento “arr[j]”for menor que “arr[i]”, ele é deslocado para sua posição correta, movendo todos os elementos entre i e j uma posição à frente. Esse processo garante que a ordem dos elementos já ordenados não seja comprometida. Após cada inserção, os índices e o valor de meio são ajustados para refletir a nova organização do vetor.

2.2.1. Polinômio de custo assintótico

Ao realizar uma avaliação do custo assintótico de tempo do algoritmo **MergeSort Alternativo**, é necessário analisar o custo de cada linha, em especial os custos dás funções “ merge_sort_in_place ”e “merge_in_place”, aqui apresentadas 6 e 7 respectivamente.

Analisando a função “ merge_sort_in_place ” 6, observa-se que alem de ter o custo de realizar as ações, como comparações e somas, está função realiza, duas chamadas recursivas, ao qual em cada chamada utiliza sempre a metade do vetor atual para a nova sub lista, ou seja, $n/2$. Assim, podemos deduzir preliminarmente uma parte da função de custo, representada como $T(n) = 2T(n/2)$. Além do custo das operações de recursivas, o código apresenta um custo base de 8 operações, e também o custo proveniente da ordenação realizada na função “merge_in_place” 7.

Ao examinar a função “merge_in_place” 7, nota-se que ela pode apresentar um comportamento problemático em vetores de tamanho grande (n). Isso ocorre devido à presença de dois laços aninhados: o laço externo “while (i <= meio \&\& j <= fim)”e o laço interno “for (int k = j; k > i; k−−)”. No pior cenário, quando o vetor está distribuído de forma aleatória ou ordenado de maneira decrescente, o algoritmo é forçado a entrar repetidamente no segundo laço, o que agrava significativamente o custo computacional. Por outro lado, em casos mais favoráveis, como quando o vetor está ordenado, o segundo laço não é executado. Nesse caso, o custo da função se limita às comparações

realizadas no laço externo, resultando em $O(n)$.

No pior cenário, no entanto, o laço interno "for"será executado $O(n)$ vezes para cada iteração do laço externo "while", que também possui $O(n)$ iterações. Isso resulta em um polinômio de custo assintótico $3n^2 + 18n + 3$.

A polinômio de custo assintótico de custo do MergeSort Alternativo, portanto, pode ser expressa como:

$$T(n) = 2T\left(\frac{n}{2}\right) + 3n^2 + 18n + 11.$$

Ao aplicar o **Teorema Mestre**, temos que $n^{\log_2 2} = n$ e $n < n^2$, dessa forma sendo aplicado o caso 3. Assim, o custo assintótico do algoritmo é quadrático:

$$O(n^2).$$

3. Avaliação Prática

Como já mencionado, foi desenvolvido um terceiro código "*ChamaMerge.cpp*", cuja função é especificar qual tipo de entrada deve ser processado em cada momento, além de realizar seis execuções consecutivas e descartar os resultados da primeira, conforme solicitado na especificação do trabalho. O tempo médio das cinco execuções restantes foi armazenado para análise futura em um arquivo CSV, utilizando a biblioteca *time* da linguagem C++.

O professor da disciplina disponibilizou quatro conjuntos de dados distintos para serem processados pelas duas implementações do MergeSort. Esses conjuntos compreendem: dados ordenados, dados ordenados de forma decrescente, dados distribuídos de forma aleatória e dados parcialmente ordenados. Cada conjunto possui 20 arquivos, com tamanhos crescentes, variando de 100 números até 2 milhões de números a serem ordenados.

Os testes realizados foram organizados nos seguintes cenários:

- As duas estratégias de ordenação aplicadas a conjuntos de dados ordenados de forma decrescente;
- As duas estratégias de ordenação aplicadas a conjuntos de dados aleatórios;
- As duas estratégias de ordenação aplicadas a conjuntos de dados ordenados de forma crescente;
- As duas estratégias de ordenação aplicadas a conjuntos de dados parcialmente ordenados de forma crescente.

Para realizar esses testes, os dados foram organizados em pastas, de acordo com sua natureza, e o código "*ChamaMerge.cpp*"foi compilado e executado em um *prompt* de comando. Todos os testes foram realizados na mesma máquina, com as seguintes configurações: processador 12th Gen Intel(R) Core(TM) i5-12450H 2.00 GHz, 8,00 GB de memória RAM (7,71 GB utilizáveis) e sistema operacional Windows 11 de 64 bits.

Os códigos que processaram dados ordenados e parcialmente ordenados foram executados até sua conclusão. Entretanto, nos casos de dados aleatórios e ordenados

de forma decrescente, a execução foi interrompida pelos autores ou devido a falhas do sistema operacional (como *tela azul*). Esses códigos permaneceram em execução por mais de seis horas, alcançando a ordenação de valores na casa dos milhões antes de serem encerrados.

4. Resultados

A partir do experimento prático realizado, foram obtidos dados referentes ao tempo de execução das duas abordagens do algoritmo MergeSort em diferentes cenários: dados ordenados, parcialmente ordenados, ordenados de forma decrescente e ordenados de forma aleatória, conforme descrito anteriormente. Para uma análise mais detalhada do desempenho dos algoritmos, cada conjunto de dados será avaliado individualmente. Essa abordagem permite examinar o comportamento de cada algoritmo em cenários distintos, identificando os fatores que influenciaram positiva ou negativamente seu desempenho.

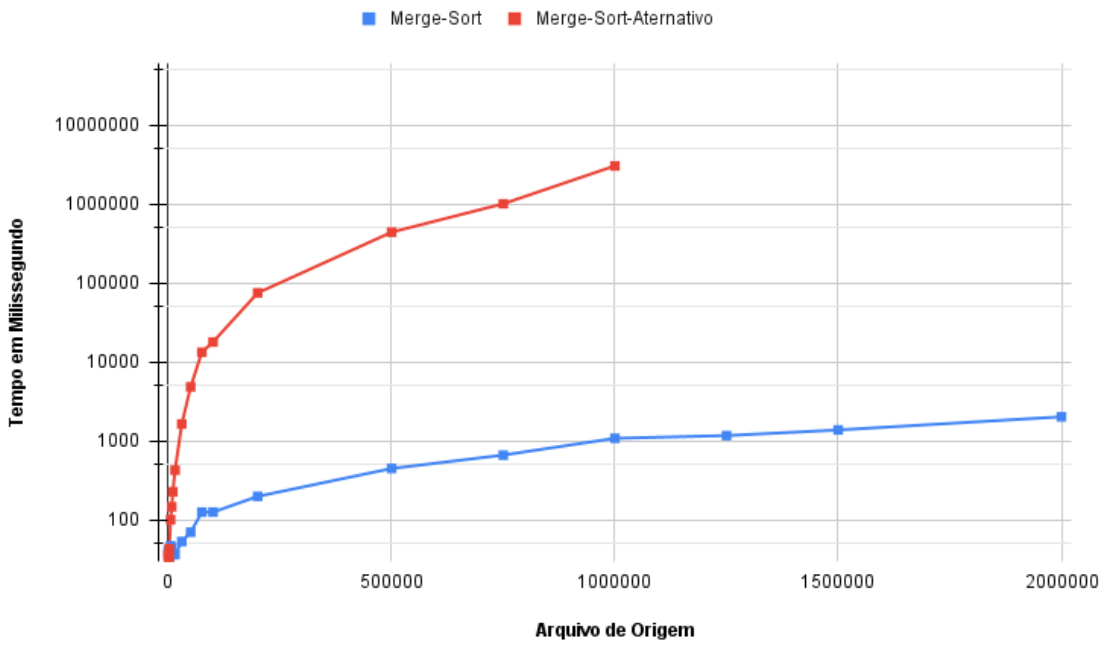
4.1. Desempenho sobre o conjunto de dados ordenados de forma decrescentes

Como já mencionado, ao todo havia vinte arquivos de cada tipo de dados, os quais apresentaram um aumento na quantidade de valores a serem ordenados, iniciando com 100 valores até 2 milhões de valores que deveriam ser ordenados de forma crescente.

A análise realizada refere-se ao seguinte teste: "As duas estratégias de ordenação aplicadas a conjuntos de dados ordenados de forma decrescente". Conforme especificado, foram aplicados os dois algoritmos sobre o conjunto de dados e coletado o tempo de execução, com o objetivo de verificar o desempenho dos algoritmos. O Gráfico presente na Figura 1 traz justamente a representação gráfica dos dados coletados. Logo na primeira análise, dois pontos chamam atenção: primeiro, o desempenho insatisfatório do MergeSort Alternativo, que apresentou um tempo de execução muito maior do que a versão clássica do MergeSort; segundo, a ausência de dados do MergeSort Alternativo para valores acima de 1 milhão de números. Isso ocorreu devido a uma decisão dos autores de não continuar a execução do algoritmo, considerando seu elevado tempo de execução. Por exemplo, com 1 milhão de dados a serem ordenados, o algoritmo apresentou um tempo médio de 3.031.945,6 milissegundos, o que equivale a 50 minutos. Considerando que o teste foi repetido 6 vezes, o tempo total superou 5 horas. Além disso, por se tratar de um algoritmo com comportamento quadrático, o tempo de execução aumentaria exponencialmente com o crescimento da quantidade de dados.

É importante destacar que o Gráfico presente na Figura 1 está representado na escala logarítmica, ou seja, expressa a evolução de um número com base em uma escala logarítmica, permitindo uma melhor visualização dos dados devido à grande diferença entre os valores.

Figura 1. Tempo de execução em milissegundos para dados ordenados de forma decrescente, com valores variando de 100 a 2 milhões de valores.



Fonte: Autoria própria

Devido ao fato de o Merge Sort Alternativo ser um algoritmo quadrático no pior caso, seu desempenho é significativamente impactado quando os elementos estão ordenados de forma decrescente. Isso ocorre porque o algoritmo precisa deslocar toda a lista para a direita sempre que encontra um valor fora da ordem correta, executando Código 7, como já discutido, esse trecho de código, torna o algoritmo quadrático, sendo ineficiente para conjuntos de dados grandes. Como é possível perceber no gráfico da Figura 1, o Merge Sort Alternativo apresenta um desempenho muito mais baixo em comparação com o Merge Sort clássico. No arquivo contendo 2 milhões de números, o Merge Sort clássico levou em média apenas 2.017,6 milissegundos (aproximadamente 2 segundos) para ordenar os dados, enquanto o Merge Sort Alternativo não foi sequer executado devido ao alto tempo de processamento necessário.

Essa superioridade do Merge Sort clássico ocorre pelo fato de que ele utiliza vetores auxiliares para armazenar os dados durante o processo de ordenação, evitando assim o alto custo de processamento. Embora o Merge Sort clássico requeira mais memória devido à necessidade de alocar esses vetores auxiliares, o custo de processamento é muito mais baixo, especialmente quando comparado ao custo exponencial do Merge Sort Alternativo.

A Tabela 1 apresenta todos os valores de execução para facilitar a visualização dos dados numéricos, permitindo observar claramente o aumento gradual do tempo de execução da versão alternativa, enquanto a versão clássica exibe um aumento muito mais controlado.

Tabela 1. Comparação de tempo de execução (em milissegundos) entre Merge-Sort e Merge-Sort Alternativo para dados decrescentes

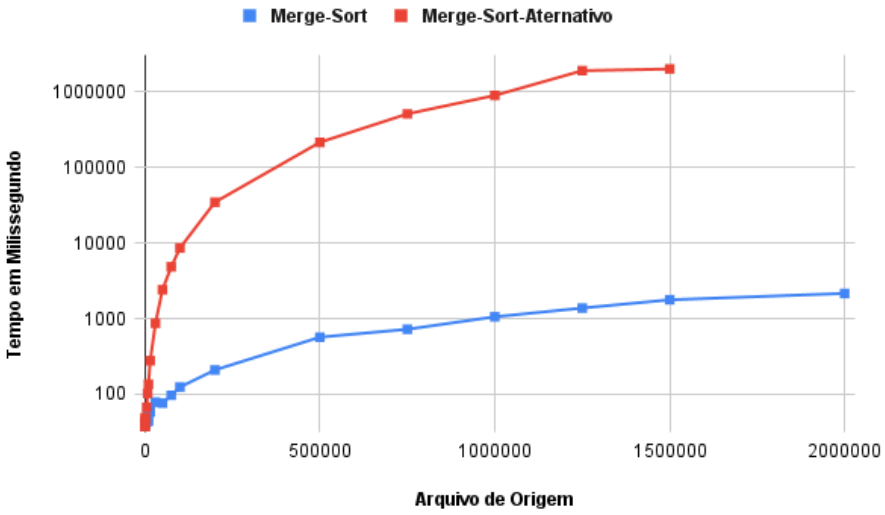
Arquivo	Merge-Sort	Merge-Sort Alternativo
d100	40,8	35,8
d200	38,2	36,4
d500	30,6	34,4
d1000	29,4	29,4
d2000	34,8	43,2
d5000	47,0	100,4
d7500	44,4	146,8
d10000	36,4	226,4
d15000	36,6	427,2
d30000	53,2	1637,0
d50000	69,8	4832,6
d75000	125,2	13272,4
d100000	125,2	17880,4
d200000	198,4	74671,2
d500000	447,4	438902,0
d750000	661,4	1003861,2
d1000000	1077,6	3031945,6
d1250000	1167,8	-
d1500000	1374,4	-
d2000000	2017,6	-

4.2. Desempenho sobre o conjunto de dados aleatórios

O segundo teste realizado teve como objetivo comparar “As duas estratégias de ordenação aplicadas a conjuntos de dados aleatórios”. Como era esperado, os resultados foram semelhantes aos obtidos com dados ordenados de forma decrescente. No entanto, houve uma melhora significativa no desempenho dos algoritmos, especialmente do Merge Sort Alternativo. Como mostrado no Gráfico da Figura 2 e na Tabela 2, foi possível executar o Merge Sort Alternativo com valores maiores, sendo apenas o arquivo com 2 milhões de números que não pôde ser processado.

Essa melhoria no desempenho do Merge Sort Alternativo é evidenciada pelo tempo de execução. No arquivo com 1 milhão de números, o Merge Sort Alternativo levou 3.031.945,6 milissegundos (aproximadamente 50 minutos) com dados ordenados de forma decrescente, enquanto com dados aleatórios, o mesmo arquivo foi ordenado em apenas 892.390,8 milissegundos, o que equivale a cerca de 14 minutos.

Figura 2. Tempo de execução em milissegundos para dados distribuídos de forma aleatória, com valores variando de 100 a 2 milhões de valores.



Fonte: Autoria própria

Tabela 2. Comparação de tempo de execução (em milissegundos) entre Merge-Sort e Merge-Sort Alternativo para dados aleatórios

Arquivo	Merge-Sort	Merge-Sort Alternativo
a100	31,6	33,2
a200	40,4	38,4
a500	43,6	48,2
a1000	42,2	42,6
a2000	42,8	49,6
a5000	43,2	67,2
a7500	50,8	102,4
a10000	44,2	136,2
a15000	58,2	278,0
a30000	78,6	868,6
a50000	76,2	2418,2
a75000	97,0	4835,8
a100000	124,4	8581,6
a200000	209,6	34617,0
a500000	569,0	214429,8
a750000	723,2	509102,6
a1000000	1057,4	892390,8
a1250000	1380,4	1897580,0
a1500000	1773,0	1997092,4
a2000000	2155,6	-

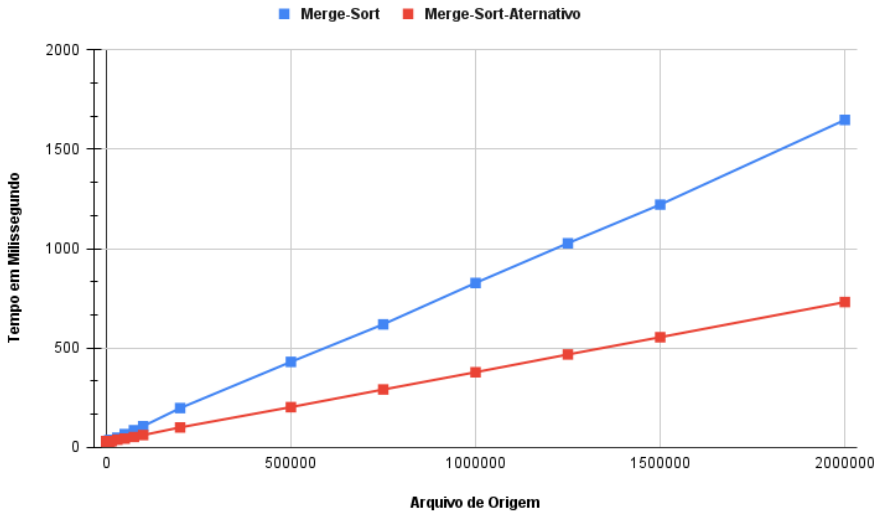
Ao analisar o comportamento dos dois algoritmos no Gráfico da Figura 1 ou os tempos apresentados na Tabela 2, é evidente a superioridade do Merge Sort clássico em relação à versão alternativa. Isso se deve, novamente, ao uso de vetores auxiliares no Merge Sort clássico, o que reduz o custo quadrático da versão alternativa. O melhor desempenho do Merge Sort Alternativo nos dados aleatórios pode ser atribuído à menor necessidade de executar o código quadrático, devido à seguinte condição: "`if (arr[i] <= arr[j])`." Em dados aleatórios, é mais provável que os valores já estejam próximos de suas posições ordenadas, reduzindo o número de comparações e deslocamentos necessários.

Por mais que ocorreu uma melhoria no desempenho do algoritmo MergeSort-Alternativo, ainda a uma grande diferença para o tempo de execução do Merge Sort clássico, por exemplo para ordenar 1 milhão e 750 mil números o MergeSort clássico levou apenas 1773 milissegundos(1,7 segundos), enquanto a versão alternativa levou 1997092,4 milissegundos(33 minutos). Dessa forma foi novamente necessário a utilização de um gráfico na escala logarítmica, para uma melhor visualização dos dados.

4.3. Desempenho sobre o conjunto de dados ordenados de forma crescente

O terceiro teste realizado, "As duas estratégias de ordenação aplicadas a conjuntos de dados ordenados de forma crescente". Neste cenário aconteceu alguns pontos diferentes dos dois teste anteriores, o primeiro é o tempo significativamente inferior, sendo possível realizar o teste sobre todos os conjuntos de dados, e inclusive o gráfico da Figura 3, que ilustra os tempos de execução, não exigiu nenhuma adequação para representar todos os valores. Outro ponto relevante é o desempenho do Merge Sort Alternativo, ao qual foi mais eficiente que o Merge Sort Clássico, apresentando um menor tempo de execução para valores maiores.

Figura 3. Tempo de execução em milissegundos para dados ordenados de forma crescente, com valores variando de 100 a 2 milhões de valores.



Fonte: Autoria própria

Como é possível observar no gráfico da Figura 3, o desempenho das duas versões do Merge Sort foi bastante satisfatório em dados ordenados, a ponto de seus tracejados se sobreporem no gráfico até o processamento de arquivos com 50 mil números. Após essa marca, ocorre um leve aumento gradual no tempo de execução da versão clássica.

Tabela 3. Comparação de tempo de execução (em milissegundos) entre Merge-Sort e Merge-Sort Alternativo para dados ordenados

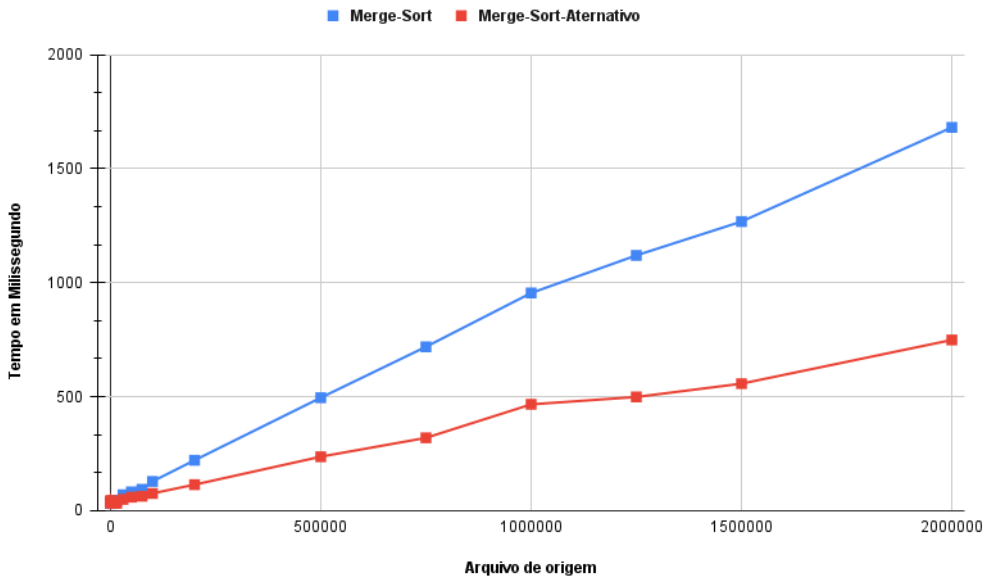
Arquivo	Merge-Sort	Merge-Sort Alternativo
o100	31	31
o200	28	29
o500	28	28
o1000	31	31
o2000	27	27
o5000	32	29
o7500	34	29
o10000	36	30
o15000	37	32
o30000	49	39
o50000	67	44
o75000	86	52
o100000	107	62
o200000	197	100
o500000	430	203
o750000	619	291
o1000000	827	378
o1250000	1027	467
o1500000	1222	554
o2000000	1648	731

Ao analisar a Tabela 3, percebe-se que os tempos de execução são semelhantes, ou até mesmo idênticos, até a marca de 50 mil números. Contudo, após esse ponto, os valores do Merge Sort Alternativo permanecem estáveis, enquanto o Merge Sort Clássico apresenta um aumento gradual. Essa diferença ocorre pelo mesmo motivo que torna a versão clássica superior em dados aleatórios e a versão alternativa menos eficiente. No caso do Merge Sort Clássico, ele sempre precisa alocar vetores auxiliares e preenchê-los com os valores, o que adiciona um tempo de execução diretamente proporcional ao tamanho do vetor de entrada, mesmo quando os dados já estão ordenados. Por outro lado, a versão alternativa do Merge Sort não precisa executar o trecho de código responsável pelo comportamento quadrático, devido à seguinte condição: "`if (arr[i] <= arr[j])`". Em dados já ordenados, essa condição é sempre verdadeira, eliminando a necessidade de deslocamentos na lista. Dessa forma, a versão alternativa deixa de ser quadrática em seu melhor caso, apresentando uma complexidade linear $O(n)$. Isso ocorre porque o algoritmo realiza apenas a "quebra"dos vetores, característica do Merge Sort, e posteriormente percorre a lista completamente para verificar os valores.

4.4. Desempenho sobre o conjunto de dados parcialmente ordenados de forma crescente

O último teste realizado foi o seguinte: “As duas estratégias de ordenação aplicadas a conjuntos de dados parcialmente ordenados de forma crescente”. Devido à semelhança dos dados com o teste de valores totalmente ordenados, era esperado um comportamento semelhante, o Gráfico da Figura 4 confirma essa expectativa, evidenciando a proximidade no comportamento das curvas das Figuras 3 e 4. Contudo, como mostrado tanto no Gráfico da Figura 4 e na Tabela 4, apesar da similaridade no padrão, houve um aumento no tempo de execução em ambos os algoritmos.

Figura 4. Tempo de execução em milissegundos para dados parcialmente ordenados de forma crescente, com valores variando de 100 a 2 milhões de valores.



Fonte: Autoria própria

O Merge Sort Alternativo novamente demonstrou superioridade, principalmente por executar a condição responsável por seu comportamento quadrático em poucas ocasiões.

Analisando a Tabela 4, nota-se que, no início, os tempos de execução de ambos os algoritmos foram semelhantes. Em cenários com conjuntos menores, o Merge Sort Alternativo chegou a ser ligeiramente inferior. Todavia, com o aumento no uso de memória, o tempo de execução do Merge Sort Clássico cresceu mais rapidamente, evidenciando a vantagem do Merge Sort Alternativo em cenários parcialmente ordenados.

Tabela 4. Comparação de tempo de execução (em milissegundos) entre Merge-Sort e Merge-Sort Alternativo para dados parcialmente ordenados

Arquivo	Merge-Sort	Merge-Sort Alternativo
po100	33	35,4
po200	39	37,0
po500	30	34,2
po1000	38	44,6
po2000	37	35,2
po5000	39	38,2
po7500	42	32,2
po10000	45	34,2
po15000	45	32,0
po30000	70	49,0
po50000	82	58,8
po75000	93	62,6
po100000	128	75,2
po200000	221	113,6
po500000	495	236,4
po750000	719	319,0
po1000000	955	466,0
po1250000	1120	498,4
po1500000	1268	556,8
po2000000	1681	748,6

5. Considerações Finais

A partir dos testes aqui apresentados concluímos que, o desempenho se tratando do tempo de execução do MergeSort Alternativo foi consideravelmente inferior ao Merge-Sort Clássico quando era necessário efetivamente a ordenação dos dados (decrescente e aleatórios), isso devido ao fato de o MergeSort Alternativo ser um algoritmo quadrático no pior caso. A diferença entre eles foi tão grande que no arquivo contendo 2 milhões de números decrescentes, o MergeSort clássico levou aproximadamente uma média de apenas 2 segundos para ordenar os dados, enquanto o MergeSort Alternativo não foi sequer executado devido ao alto tempo de processamento necessário. Contudo, devemos levar em consideração que o custo de memória no MergeSort Clássico foi muito elevado, isso devido a necessidade de alocação dos vetores auxiliares que o algoritmo utiliza.

Para dados aleatórios ambos os algoritmos apresentaram uma melhora em relação ao decrescente, como já era esperado, porém o MergeSort Alternativo ainda permaneceu muito inferior ao MergeSort Clássico chegando a não ser executado em tempo hábil para conjuntos de 2 milhões de números.

Em relação ao conjunto de dados ordenado, tivemos pela primeira vez o Merge-Sort Alternativo possuindo resultados melhores em relação ao MergeSort Clássico, Essa diferença ocorre pois o MergeSort Clássico sempre precisa alocar vetores auxiliares e preenchê-los com os valores, mesmo quando os dados já estão ordenados. Por outro lado, a

versão alternativa do MergeSort não precisa executar o trecho de código responsável pelo comportamento quadrático, eliminando a necessidade de deslocamentos na lista.

Por fim os testes sobre conjuntos de dados parcialmente ordenados nos mostraram que o MergeSort Alternativo novamente foi superior ao MergeSort Clássico, principalmente por ter que executar a condição responsável por seu comportamento quadrático em poucas ocasiões. Porém é importante salientar que em cenários com conjuntos menores, o MergeSort Alternativo chegou a ser ligeiramente inferior, mas com o aumento no uso de memória, o tempo de execução do MergeSort Clássico cresceu mais rapidamente, evidenciando a vantagem do MergeSort Alternativo em cenários parcialmente ordenados.

Com isso podemos ter um panorama geral de que o MergeSort Alternativo é mais interessante em comparação ao MergeSort Clássico quando se trata de dados já ordenados ou parcialmente ordenados, pois seu trecho de código onde sua complexidade é quadrática passa a ser executada nenhuma ou poucas vezes, porém para dados aleatórios ou decrescentes o MergeSort Clássico leva uma vantagem esmagadora em relação ao MergeSort Alternativo, isso devido ao fato de que o MergeSort Clássico utiliza vetores auxiliares para armazenar os dados durante o processo de ordenação, evitando assim o alto custo de processamento, porém é importante salientar que o MergeSort clássico requer mais memória devido à necessidade de alocar esses vetores auxiliares.

Referências

Akhter, N., Idrees, M., and Rehman, F. (2016). Sorting algorithms - a comparative study. *Journal of Computer Science and Information Security*, 14(14):1–7.

Brunet, J. (2019). Ordenação por comparação: Merge sort. Acessado em: 9 de novembro de 2024.

Cormen, T., Leiserson, C., Rivest, R., and Stein, C. (2009). *Introduction to Algorithms*, volume 3. MIT Press (MA).

for Geeks, G. (2023). In-place merge sort. Acessado em :21 de novembro de 2024.

Gul, M., Amin, N., and Suliman, M. (2015). An analytical comparison of different sorting algorithms in data structure. *International Journal of Advanced Research in Computer Science and Software Engineering*.