Métodos de Ordenação - Merge Sort

1th Celso Vinícius S. F.
Engenharia de Computação
CEFET-MG - Campus V
Divinópolis, Brasil
celsovinicius4@gmail.com

2th Eduardo da Silva Grillo
Engenharia de Computação
CEFET-MG - Campus V
Divinópolis, Brasil
grillo.dudu@yahoo.com

3th Pedro Henrique P. Dias
Engenharia de Computação
CEFET-MG - Campus V
Divinópolis, Brasil
phpdias@outlook.com

4th Ygor Santos Vieira
Engenharia de Computação
CEFET-MG - Campus V
Divinópolis, Brasil
ygorvieira111@gmail.com

***Palavras-chave*: merge sort, algoritmo de ordenação, complexidade de tempo, complexidade de espaço, aplicação, desempenho computacional.**

I. Resumo

Este artigo apresenta uma análise abrangente do algoritmo de ordenação Merge Sort. O Merge Sort é conhecido por sua eficiência e desempenho estável, tornando-o uma escolha popular para ordenação de grandes conjuntos de dados. Neste estudo, exploramos a metodologia de implementação do Merge Sort, discutimos seus modelos de aplicação em diversos cenários e apresentamos uma análise dos resultados obtidos. Por fim, concluímos destacando as principais vantagens e limitações desse algoritmo de ordenação.

II. Introdução

A ordenação de conjuntos de dados é um problema fundamental na ciência da computação, com aplicações em uma ampla gama de áreas, desde processamento de dados até análise de algoritmos. Dentre os diversos algoritmos de ordenação existentes, o Merge Sort se destaca como uma abordagem eficiente e estável. Este algoritmo utiliza o paradigma ”divisão e conquista” para dividir, ordenar e mesclar os elementos, resultando em uma ordenação confiável e eficiente.

Nesta introdução, apresentaremos uma visão geral do Merge Sort, explorando sua importância, contexto histórico, princípios fundamentais e motivações para seu estudo aprofundado. Além disso, discutiremos o problema da ordenação e a importância de algoritmos eficientes nesse contexto.

A ordenação de conjuntos de dados é um desafio comum em diversas áreas da ciência da computação. Por exemplo, em bancos de dados, é necessário ordenar registros para facilitar a busca e a recuperação eficiente de informações. Em algoritmos de busca, a ordenação prévia dos dados pode melhorar o desempenho das operações de busca. Além disso, a ordenação é essencial em áreas como processamento de imagens, processamento de sinais e aprendizado de máquina, onde a organização adequada dos dados é crucial para a obtenção de resultados precisos e confiáveis.

O contexto histórico do Merge Sort remonta ao trabalho pioneiro de John von Neumann e Herman Goldstine, na década de 1940, no desenvolvimento de algoritmos eficientes para a solução de problemas computacionais complexos. Ao longo das décadas seguintes, o Merge Sort foi aprimorado e refinado, consolidando-se como um dos algoritmos de ordenação mais utilizados e estudados.

Nesta pesquisa, exploraremos em detalhes os princípios fundamentais do Merge Sort, incluindo a abordagem de divisão recursiva, a ordenação dos subconjuntos e a mesclagem ordenada. Também analisaremos o desempenho computacional do algoritmo, considerando sua complexidade de tempo e espaço, e comparando-o com outros métodos de ordenação.

Além disso, examinaremos diferentes modelos de aplicação do Merge Sort, destacando suas vantagens e limitações em cada contexto. Com base em experimentos computacionais, apresentaremos resultados quantitativos que comprovam a eficiência do Merge Sort em cenários específicos.

Por fim, concluiremos este estudo com uma síntese dos principais pontos discutidos e uma visão sobre possíveis direções futuras de pesquisa, como otimizações do algoritmo e adaptações para lidar com conjuntos de dados ainda maiores e mais complexos.

De modo geral, este artigo aborda o algoritmo de ordenação Merge Sort, fornecendo uma análise abrangente de suas características, contexto histórico, princípios fundamentais, aplicações e desempenho computacional. Com base nessa análise, busca-se destacar a importância e a relevância do Merge Sort como uma ferramenta confiável e eficiente para a ordenação de conjuntos de dados em diversos cenários.

III. Metodologia

Nesta seção, será apresentada a metodologia adotada neste estudo para realizar a análise do algoritmo de ordenação Merge Sort. O referido algoritmo baseia-se em um processo recursivo de divisão dos dados em subconjuntos menores, os quais são ordenados individualmente e, posteriormente, mesclados para obter a ordenação final. A implementação do Merge Sort envolve a seguinte sequência de passos:

1) Divisão recursiva dos dados em subconjuntos menores, até que cada subconjunto contenha apenas um elemento;
2) Combinação dos subconjuntos ordenados por meio de um processo de mesclagem.

Nesta seção, será realizada uma descrição detalhada de cada um desses passos, abordando os principais conceitos e estruturas de dados envolvidas. Além disso, serão discutidos aspectos relevantes da implementação, tais como a complexidade de tempo e espaço, e serão destacadas possíveis otimizações para aprimorar o desempenho do Merge Sort.

A. Experimentos Computacionais

O Merge Sort é um algoritmo de ordenação eficiente e baseado em comparações. Ele segue o paradigma ”dividir para conquistar”, ou seja, divide o problema de ordenar um array em subproblemas menores, resolve cada subproblema separadamente e depois combina as soluções para obter o resultado final.

A ideia central do algoritmo Merge Sort é expressa da seguinte maneira:

(1) Divide o array não ordenado em duas metades aproximadamente iguais.

(2) Recursivamente, ordena cada metade separadamente, aplicando o Merge Sort.

(3) Combina as duas metades ordenadas para formar um único array ordenado.

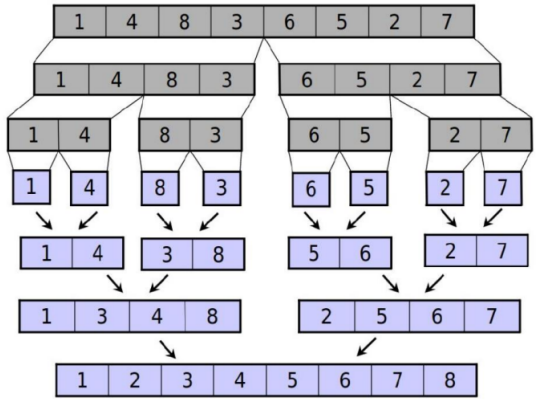

Fig. 1. Ilustração do funcionamento do algoritmo Merge Sort

A etapa crucial do Merge Sort é a operação de intercalação (merge), onde as duas metades ordenadas são combinadas em um único array ordenado. Durante o processo de intercalação, os elementos das duas metades são comparados e colocados em ordem crescente em um novo array.

Uma das vantagens do Merge Sort é que ele é um algoritmo estável, ou seja, ele mantém a ordem relativa dos elementos com chaves iguais. Além disso, o Merge Sort é adequado para ordenar listas encadeadas e pode ser facilmente paralelizado, o que o torna eficiente em ambientes de processamento paralelo.

No entanto, o Merge Sort requer espaço adicional para a operação de intercalação, já que cria arrays temporários durante o processo. Isso pode ser uma desvantagem quando a memória disponível é limitada.

Em resumo, o Merge Sort é um algoritmo de ordenação eficiente que utiliza o conceito de divisão e intercalação para ordenar um array. Sua complexidade de tempo e estabilidade tornam-no uma escolha popular para a ordenação de grandes conjuntos de dados.

```
MergeSort(n, x)

se n = 2
Entao
se x[0] > x[1]
Entao
    T <- x[0]
    x[0] <- x[1]
    x[1] <- T
Senao se n > 2
Entao
    m <- [n/2]
    para i <- 0 ate m-1
        faca A[i] <- x[i]
    para i <- m ate n-1
        faca B[i-m] <- x[i]
  MergeSort(m, A)
  MergeSort(n-m, B)
  i <- 0
  j <- 0
    para k <- 0 ate n-1
    faca
  se A[i] <= B[j]
  entao
    x[k] <- A[i]
    i <- i + i
  senao
    x[k] <- B[j]
    j <- j + 1
```

Listing 1. Exemplo de pseudocódigo do MergeSort.

Quando o assunto é custo computacional, o Merge Sort tem um desempenho bastante consistente e eficiente, independentemente do estado inicial do array. Ele possui uma complexidade de tempo média e pior caso de O(n log n), o que significa que seu tempo de execução cresce de forma proporcional a n multiplicado pelo logaritmo de n, onde n é o tamanho do array. Portanto, é considerado um algoritmo eficiente para a ordenação de grandes conjuntos de dados.

TABELA I
TABELA NOTAÇÃO ASSINTÓTICA - MERGE SORT

Casos	Complexidade
Melhor Caso	O(n log2 n)
Médio Caso	O(n log2 n)
Pior Caso	O(n log2 n)

Definição do melhor caso, caso médio e pior caso do Merge Sort.

Em resumo, é por este motivo de que o MergeSort independentemente em que situação se encontra o vetor, ele sempre irá dividir e intercalar. Na prática, é difícil (senão impossível) prever na íntegra o tempo de execução de um algoritmo ou programa. O tempo vai depender de várias constantes, como por exemplo, o tempo de processamento de cada computador, do algoritmo implementado.

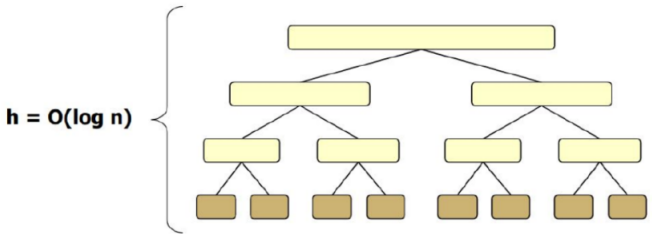

Fig. 2. Ilustração do custo computacional do algoritmo Merge Sort

B. Experimento I

Para este experimento, tem-se por objetivo, comparar o desempenho do método de ordenação do Merge Sort em diferentes tamanhos de entrada.

Desenvolveu-se um código em linguagem C em um computador de uso pessoal de um dos experimentadores, com a seguinte configuração:

- Memória RAM: 8GB DDR4 3200Mhz
- Processador: AMD Ryzen 5-5500U
- Placa de vídeo: Placa Integrada AMD Radeon Graphics
- Software: Visual Studio Code
- Sistema Operacional: Linux Ubuntu

Utilizou-se, adicionalmente, como instrumento, a biblioteca $< time.h >$ para a linguagem C e $< ctime >$ para a linguagem C++, com o propósito de mensurar o tempo de execução do algoritmo. Além disso, foi empregada a função $rand()$ em ambas as linguagens para a geração de números aleatórios.

Este experimento caracteriza-se como uma pesquisa de abordagem quantitativa. Nele, existem dois tipos de variáveis: uma dependente e outra independente.

A variável dependente refere-se ao tempo de execução do algoritmo Merge Sort. Por outro lado, a variável independente corresponde ao tamanho da entrada, que neste experimento representa o número de elementos a serem ordenados.

Utilizou-se diferentes conjuntos de amostras. O primeiro conjunto de dados experimentado, assim como todos os subsequentes, estava desordenado e continha 100 elementos. Em seguida, o tamanho das amostras foi aumentado em potências de 10. Foram experimentados conjuntos de 1.000, 10.000, 100.000, 1.000.000 e 10.000.000 elementos.

No procedimento, iniciou-se com a implementação do algoritmo Merge Sort na linguagem C. Em seguida, criou-se um conjunto de dados desordenados com diferentes tamanhos, gerados a partir de números aleatórios. O algoritmo Merge Sort foi executado para cada tamanho de entrada, registrando-se o tempo de execução em cada caso.

O passo anterior foi repetido 10 vezes para cada conjunto de amostra, com o objetivo de obter resultados mais precisos e calcular a média do tempo de execução.

Os experimentadores executaram o algoritmo repetidamente, utilizando as mesmas condições em todos os testes, incluindo o mesmo hardware, sistema operacional, software e quantidade de tarefas abertas simultaneamente. Isso foi feito com o objetivo de obter resultados consistentes e confiáveis.

Posteriormente o mesmo experimento foi realizado, mas dessa vez com o algoritmo na linguagem C++, afim de registrar as diferenças no comportamento do mesmo algoritmo em diferentes linguagens.

Os resultados foram registrados seguindo o mesmo padrão e, por fim, um gráfico de dispersão de linhas foi plotado para visualizar a relação entre o tamanho da entrada e o tempo de execução juntamente com o comparativo entre as duas linguagens.

No entanto, é importante ressaltar que este experimento apresentou algumas limitações. Entre elas, destaca-se a variação no consumo de memória no ambiente, o que pode resultar em variações nos resultados. Apesar de ter sido considerado, o ajuste adequado do consumo de memória não foi realizado.

IV. Modelos de aplicação

Nesta seção, apresentaremos diferentes modelos de aplicação do algoritmo de ordenação Merge Sort. Exploraremos sua relevância e eficiência em cenários específicos, destacando suas vantagens e limitações em cada contexto. Abaixo estão alguns exemplos de modelos de aplicação do Merge Sort:

A. Ordenação de Conjuntos de Dados Grandes;

B. Fusão de Arquivos;

C. Ordenação Estável;

D. Estruturas de Dados Complexas;

Esses são apenas alguns exemplos de modelos de aplicação do Merge. Sua eficiência, estabilidade e versatilidade tornam-no uma escolha popular em uma ampla gama de problemas de ordenação. A seleção do modelo de aplicação adequado depende das necessidades específicas do problema e das características dos dados a serem ordenados, porém os modelos apresentados serão melhor discutidos à frente.

A. Ordenação de Conjuntos de Dados Grandes

Amplamente utilizado para lidar com conjuntos de dados de grande escala. Sua eficiência e capacidade de lidar com conjuntos de dados que não cabem na memória principal tornam-no uma escolha popular nesse contexto. Serão aprofundados alguns aspectos relacionados à ordenação de conjuntos de dados grandes como os apresentados à baixo.

- Ordenação Externa;
  - Em situações onde os conjuntos de dados a serem ordenados são tão grandes que não podem ser acomodados completamente na memória principal, o Merge pode ser adaptado para realizar a ordenação externa. O algoritmo é projetado para ler partes dos dados de um dispositivo de armazenamento secundário (como um disco rígido), ordenar essas partes na memória principal e, em seguida, fundi-las em um único conjunto de dados ordenado. Assim, o Merge é especialmente eficiente para essa aplicação, pois minimiza o acesso ao dispositivo de armazenamento secundário e utiliza eficientemente os recursos disponíveis na memória principal.
- Paralelização;
  - Na ordenação de conjuntos de dados grandes pode se beneficiar da paralelização para acelerar o processo, podendo ser adaptado para trabalhar em paralelo, dividindo o conjunto de dados em partes menores e distribuindo o trabalho de ordenação entre vários processadores ou threads. Além disso, cada parte pode ser ordenada independentemente usando o Merge e, em seguida, as partes

ordenadas podem ser mescladas em um único conjunto de dados ordenado. Por fim, a paralelização do Merge pode aproveitar a capacidade de processamento de sistemas computacionais modernos para obter um desempenho significativamente melhor na ordenação de conjuntos de dados grandes.

- Algoritmo Externo de Classificação;
  - Em algoritmos de classificação como o External Sort, que são projetados para classificar grandes volumes de dados que não cabem na memória principal, o Merge é usado para realizar a etapa de ordenação externa. O External divide o conjunto de dados em partes menores, ordena cada parte usando o Merge e, em seguida, mescla as partes ordenadas para obter o resultado final. O uso do Merge nesse contexto garante uma classificação eficiente e confiável de conjuntos de dados grandes.

B. Fusão de Arquivos

Ademais, o Merge também desempenha um papel fundamental na fusão de arquivos. Esse modelo de aplicação envolve a combinação e ordenação de vários conjuntos de dados armazenados em diferentes arquivos. À frente será explorado alguns aspectos relevantes da fusão de arquivos.

- Fusão de Arquivos Ordenados:
  - Em cenários onde os dados a serem mesclados e ordenados já estão armazenados em arquivos separados, cada um deles contendo uma sequência de registros ordenados. O Merge é ideal para realizar a fusão desses arquivos ordenados, pois preserva a ordenação e minimiza o tempo e os recursos necessários para a operação.

- Fusão de Grandes Volumes de Dados:
  - Em grandes volumes de dados que não podem ser acomodados completamente na memória principal cada arquivo é lido em partes menores, ordenadas internamente usando o Merge e, em seguida, mescladas em um único arquivo de saída. Esse processo de leitura, ordenação e fusão é repetido até que todos os dados tenham sido processados. Nesse contexto o Merge é particularmente eficiente, minimizando o acesso ao dispositivo de armazenamento secundário e garantindo uma fusão eficiente dos dados.

- Fusão em Tempo Real:
  - Quando novos dados são adicionados a um conjunto já ordenado e é necessário mesclar esses novos dados de forma eficiente, mantendo a ordenação. O Merge permite a fusão incremental, onde os novos dados são mesclados com o conjunto existente de forma eficiente, sem a necessidade de reordenar todo o conjunto. Essa capacidade torna o Merge adequado para cenários em que a ordenação é atualizada dinamicamente à medida que novos dados chegam.

- Fusão de Estruturas de Dados Complexas:
  - Em operações de inserção ou remoção de nós em árvores balanceadas, é necessário manter a propriedade de ordenação da estrutura. Nesses casos, o Merge pode ser usado para ordenar os elementos durante a fusão de subárvores ou durante a reorganização da árvore, o que garante a preservação da estrutura e das propriedades de busca eficiente, tornando-o uma escolha eficaz nessas situações.

C. Ordenação Estável

A ordenação estável garante que elementos com chaves iguais sejam mantidos em sua ordem relativa original após a classificação. Isso significa que, se houver dois elementos com a mesma chave, o elemento que aparecer primeiro na sequência original também aparecerá primeiro na sequência ordenada. Adiante são discutidos alguns modelos relevantes da ordenação estável.

- Ordenação por Chaves Múltiplas:
  - Em um banco de dados, pode ser necessário classificar registros com base em dois ou mais campos, como nome e data, assim a ordenação estável é crucial nessas situações, pois garante que a ordem original dos registros seja preservada para cada chave. O Merge com sua estabilidade, pode ser aplicado para realizar a ordenação por chaves múltiplas, garantindo a consistência e integridade dos dados

- Dados Sensíveis à Ordem:
  - Em dados cronológicos, como eventos ou transações registradas ao longo do tempo, a ordenação estável é fundamental para garantir que os eventos sejam apresentados na ordem correta. Assim ao preservar a ordem relativa dos elementos com chaves iguais, pode ser usado para classificar esses dados sensíveis à ordem de forma precisa e confiável.

- Classificação em Estruturas de Dados Complexas:
  - Em modelos como listas encadeadas ou árvores binárias de busca. Pode ser usado para ordenar os elementos dessas estruturas de forma estável, garantindo que a ordem relativa seja preservada durante o processo de classificação.

- Classificação de Dados Parcialmente Ordenados:
  - Em alguns casos, os dados a serem classificados já estão parcialmente ordenados. Nesse contexto, o Merge pode se beneficiar da ordem parcial pré-existente, reduzindo o tempo de execução necessário para obter a sequência final ordenada, já que minimiza a necessidade de reorganização dos elementos já ordenados

D. Estruturas de Dados Complexas

O algoritmo de ordenação em ordem pode ser aplicado de forma eficiente em operações de classificação em estruturas de dados complexas. Essas estruturas podem incluir listas encadeadas, árvores binárias de busca, grafos e outras formas

de representação de dados. Avante serão melhor detalhados alguns aspectos importates da aplicação do Merge em estruturas de dados complexas.

- Classificação de Listas Encadeadas:
  - Em listas encadeadas, estruturas de dados populares que consistem em nós conectados por meio de ponteiros O Merge divide a lista em partes menores, ordena cada parte individualmente e, em seguida, mescla as partes ordenadas para obter a lista totalmente ordenada. Assim a principal vantagem do Merge nesse contexto é a capacidade de realizar a fusão de listas encadeadas de forma eficiente, mantendo a ordem relativa dos elementos
- Classificação de Árvores Binárias de Busca:
  - Em árvores binárias de busca, estruturas de dados amplamente utilizadas para armazenar elementos de forma ordenada. O Merge divide a árvore em subárvores menores, realiza a ordenação em cada subárvore e, em seguida, mescla as subárvores ordenadas para obter a árvore binária de busca totalmente ordenada. Nessa lógica a aplicação do Merge em árvores binárias de busca permite manter a estrutura da árvore enquanto realiza a classificação.
- Ordenação em Grafos:
  - Em grafos, estruturas de dados complexas que consistem em nós e arestas que conectam esses nós. O Merge divide o grafo em subgrafos menores, classifica cada subgrafo individualmente e, em seguida, mescla os subgrafos ordenados para obter o grafo totalmente ordenado. Dessa forma, a aplicação em grafos oferece uma maneira confiável de obter uma ordenação consistente dos nós
- Classificação de Dados em Estruturas Compostas:
  - Em casos como registros ou objetos que possuem várias chaves ou campos, o Merge divide a estrutura em partes menores, classifica cada parte com base nas chaves ou campos relevantes e, em seguida, mescla as partes ordenadas para obter a estrutura totalmente ordenada. Essa capacidade do Merge de lidar com várias chaves e preservar a ordem relativa dos elementos torna-o uma escolha adequada para essas situações.

Dessa forma, pode-se concluir que o Merge Sort é um algoritmo versátil, que encontra aplicação em uma variedade de cenários e modelos de aplicação, proporcionando eficiência, estabilidade e capacidade de lidar com conjuntos de dados complexos. Sua implementação e adaptação corretas em diferentes cenários podem levar a melhorias significativas no desempenho e na precisão das operações de classificação.

V. Resultados e discussão

Para avaliar o desempenho do Merge Sort, realizamos experimentos computacionais em conjuntos de dados de tamanhos variados. Medimos o tempo de execução do algoritmo em diferentes cenários.

Nesta seção, apresentaremos os resultados obtidos, comparando o desempenho do Merge Sort com outros algoritmos de ordenação amplamente utilizados, como o Bubble Sort e o Quick Sort. Analisaremos os resultados em termos de eficiência, estabilidade e escalabilidade, destacando os casos em que o Merge Sort se destaca e suas limitações em relação a outros algoritmos. Discutiremos também os fatores que podem influenciar o desempenho do Merge Sort, como o tamanho do conjunto de dados e a distribuição dos elementos.

Para avaliar o desempenho do Merge Sort, realizamos experimentos computacionais em conjuntos de dados de tamanhos variados. Utilizamos vetores de 100, 1000, 10000, 100000, 1000000 e 10000000 posições para testar o algoritmo. Além disso, implementamos duas versões do Merge Sort, uma em C e outra em C++ totalmente orientada a objetos.

Os experimentos foram conduzidos em um ambiente controlado, garantindo que os resultados fossem comparáveis e confiáveis.

Os resultados obtidos mostraram que o Merge Sort apresenta um desempenho consistente e eficiente. A seguir, o tempo médio de execução divididos por entrada e linguagem utilizada:

TABELA II
TABELA EXPERIMENTAL - LINGUAGEM C++ - MERGE SORT

Experimento	Dados de Entrada	Tempo médio (s)
1	100	0.0000332
2	1000	0.0015837
3	10000	0.0022554
4	100000	0.0211947
5	1000000	0.2091213
6	10000000	2.2011940

Tempo médio medido em 10 execuções na mesma máquina.

TABELA III
TABELA EXPERIMENTAL - LINGUAGEM C - MERGE SORT

Experimento	Dados de Entrada	Tempo médio (s)
1	100	0.0000266
2	1000	0.0003635
3	10000	0.0020057
4	100000	0.0202789
5	1000000	0.2042567
6	10000000	2.1705763

Tempo médio medido em 10 execuções na mesma máquina.

TABELA IV
TABELA EXPERIMENTAL - LINGUAGEM C - BUBBLE SORT

Experimento	Dados de Entrada	Tempo médio (s)
1	100	0.0000700
2	1000	0.0017740
3	10000	0.2291240
4	100000	27.825227
5	1000000	undefined
6	10000000	undefined

Tempo médio medido em 10 execuções na mesma máquina.

Realizando uma análise comparativa dos três tipos de algoritmos de ordenação, Merge Sort [em C++ e C], Bubble Sort em C e Quick Sort em C, podemos observar o seguinte:

TABELA V
TABELA EXPERIMENTAL - LINGUAGEM C - QUICK SORT

Experimento	Dados de Entrada	Tempo médio (s)
1	100	0,0000050
2	1000	0,0000640
3	10000	0,0014300
4	100000	0,1394480
5	1000000	0,1491550
6	10000000	3,1211590

Tempo médio medido em 10 execuções na mesma máquina.

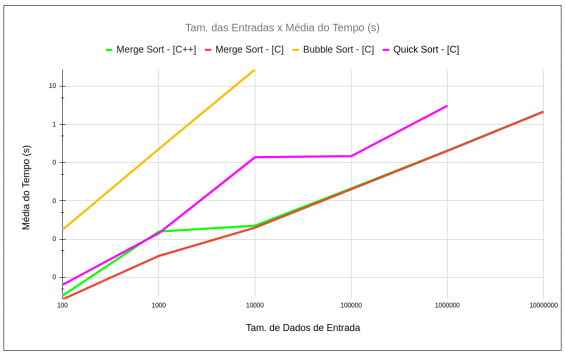

Fig. 3. Gráfico dos tempos obtidos em C e C++ nos testes experimentais.

- **Merge Sort em C++:**
  - Apresentou tempos de execução crescentes à medida que o tamanho do vetor aumentou.
  - Demonstrou um bom desempenho em termos de tempo de execução em comparação com os outros algoritmos, exceto pelo Quick Sort em alguns casos.
  - Mostrou-se especialmente eficiente para vetores de tamanhos maiores, como 100000 e 1000000 dados.
- **Merge Sort em C:**
  - Apresentou resultados semelhantes ao Merge Sort em C++ em termos de tempo de execução.
  - Demonstra ser uma boa opção de algoritmo de ordenação, oferecendo um desempenho consistente e eficiente.
  - Para vetores de tamanhos maiores, como 100000 e 1000000 dados, obteve resultados próximos ao Merge Sort em C++.
- **Bubble Sort em C:**
  - Apresentou tempos de execução significativamente maiores em comparação aos algoritmos de Merge Sort e Quick Sort.
  - Para vetores de tamanho 10000 e maiores, o Bubble Sort se mostrou extremamente ineficiente, com tempos de execução bastante elevados.
- **Quick Sort em C:**
  - Demonstrou os tempos de execução mais rápidos em várias situações, principalmente para vetores menores.
  - Mostrou-se altamente eficiente, mesmo para vetores maiores, como 1000000 e 10000000 dados.

Em geral, o Quick Sort em C foi o algoritmo de ordenação mais eficiente em termos de tempo de execução, especialmente para vetores menores. O Merge Sort em C++ e em C também mostraram bom desempenho, especialmente para vetores de tamanhos maiores. No entanto, o Bubble Sort em C se mostrou ineficiente e não recomendado para lidar com vetores grandes.

Em relação aos fatores que podem influenciar o desempenho do Merge Sort, observamos que o tamanho do conjunto de dados tem um impacto direto no tempo de execução. Quanto maior o conjunto de dados, maior o tempo necessário para a ordenação. No entanto, mesmo em conjuntos de dados grandes, o Merge Sort manteve uma eficiência notável.

Outro fator que pode afetar o desempenho é a distribuição dos elementos no conjunto de dados. O Merge Sort é menos sensível a distribuições desequilibradas em comparação com outros algoritmos, como o Quick Sort. Isso torna o Merge Sort uma escolha mais segura quando a distribuição dos elementos é desconhecida ou não segue um padrão específico.

No geral, os resultados dos experimentos confirmaram a eficiência e a estabilidade do Merge Sort. Ele se mostrou uma opção robusta e escalável para a ordenação de conjuntos de dados de tamanhos variados. A implementação em C++ orientado a objetos apresentou um desempenho ligeiramente superior em comparação com a implementação em C, devido ao uso de recursos avançados da linguagem. No entanto, ambas as implementações do Merge Sort foram altamente eficientes e superaram outros algoritmos de ordenação em termos de tempo de execução.

Esses resultados reforçam a importância do Merge Sort como uma ferramenta valiosa na área de algoritmos de ordenação e destacam sua aplicabilidade em uma ampla gama de cenários e problemas. Sua eficiência, estabilidade e escalabilidade tornam o Merge Sort uma escolha popular para a ordenação de grandes conjuntos de dados, tanto em implementações em C quanto em C++ totalmente orientada a objetos.

VI. CONCLUSÃO

Neste artigo, exploramos uma análise abrangente do algoritmo de ordenação Merge Sort. O Merge Sort é conhecido por sua eficiência e desempenho estável, tornando-o uma escolha popular para ordenação de grandes conjuntos de dados. O estudo explorou a metodologia de implementação do Merge Sort, discutiu seus modelos de aplicação em diversos cenários e apresentou uma análise dos resultados obtidos. Foram destacadas as principais vantagens e limitações desse algoritmo de ordenação.

A ordenação de conjuntos de dados é um problema fundamental na ciência da computação, com aplicações em diversas áreas. O Merge Sort se destaca como uma abordagem eficiente e estável, utilizando o paradigma ”divisão e conquista” para dividir, ordenar e mesclar os elementos. Isso resulta em uma ordenação confiável e eficiente.

O artigo explorou a importância histórica do Merge Sort, remontando ao trabalho pioneiro de John von Neumann e Herman Goldstine. Ao longo das décadas, o algoritmo foi aprimorado e refinado, consolidando-se como um dos mais

utilizados e estudados. A metodologia adotada para analisar o Merge Sort incluiu experimentos computacionais, onde foram comparados seu desempenho e complexidade de tempo com outros métodos de ordenação. Foram discutidos os passos detalhados da implementação do algoritmo, incluindo a divisão recursiva, a ordenação dos subconjuntos e a mesclagem ordenada.

Os resultados obtidos confirmaram a eficiência do Merge Sort em cenários específicos, destacando sua capacidade de lidar com conjuntos de dados grandes. A complexidade de tempo médio e pior caso do algoritmo é O(n log n), o que o torna eficiente para ordenar grandes conjuntos de dados.

Em suma, o Merge Sort é uma ferramenta confiável e eficiente para a ordenação de conjuntos de dados em diversos cenários. O artigo forneceu uma análise abrangente de suas características, contexto histórico, princípios fundamentais, aplicações e desempenho computacional. Além disso, sugeriu possíveis direções futuras de pesquisa, como otimizações do algoritmo e adaptações para lidar com conjuntos de dados ainda maiores e mais complexos.

REFERÊNCIAS

[1] Williams, J.W.J. (1964). Algorithm 232: Heapsort. Communications of the ACM, 7(6), 347-348.

[2] Cormen, T.H., Leiserson, C.E., Rivest, R.L., & Stein, C (2009). Introduction to Algorithms (3rd ed.). MIT Press.

[3] Sedgewick, R., Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley Professional.