Métodos de Ordenação - Merge Sort

1st Maíra Beatriz de Almeida Lacerda
Engenharia de Computação
CEFET-MG - Campus V
Divinópolis, Brasil
mairaallacerda@gmail.com

2nd Maria Eduarda Teixeira Souza
Engenharia de Computação
CEFET-MG - Campus V
Divinópolis, Brasil
dudateixeirasouza@gmail.com

3rd Sergio Henrique Quedas Ramos
Engenharia de Computação
CEFET-MG - Campus V
Divinópolis, Brasil
sergiohenriquequedasramos@gmail.com

***Abstract*—O Merge Sort é um método de ordenação criado por Von Neeumann em 1945, bastante utilizado em implementações que buscam um tempo de execução constante. Isto posto, este artigo busca explicar, as vantagens e desvantagens do Merge Sort quando colocado em diferentes ambientes, tendo em vista seu problema relacionado a memoria. Além de explicar a complexidade do algoritmo, dito isso sua função assintótica e seu desempenho para diferentes tipos de vetores de entrada.**

Para a realização dos testes foram utilizados diferentes tipos de linguagens compiladas e interpretadas, sendo elas: C, C++, C#, Rust, Java, JavaScript, PHP e Python. Simultaneamente a implementação da linguagem e seus compiladores, foram plotados gráficos e funções para salvar o tempo de execução de cada processo, para que dessa forma pudesse comprovar a teoria por trás do método e além disso compara-los e perceber as diferenças das linguagens de programação utilizadas e dos arquivos de entrada. Assim, foi realizada uma media de 10 execuções para cada conjunto de dados de entrada e foram apresentados em tabelas e gráficos para facilitar a visualização e a comparação dos dados.

Os achados sugerem que, embora o Merge Sort permaneça uma técnica de ordenação robusta e confiável, seu desempenho pode variar substancialmente dependendo da linguagem de programação utilizada, assim como o caso em que for utilizado.

***Index Terms*—Merge Sort, Complexidade de Tempo, Complexidade de Espaço, Análise de Algoritmos, Linguagens de Programação**

I. Introdução

Um algoritmo é um procedimento bem definido que transforma valores de entrada em valores de saída, resolvendo um problema específico. Ele descreve como alcançar a relação desejada entre entrada e saída. A ordenação de dados é essencial na ciência da computação, impactando o desempenho de muitas aplicações e algoritmos. Ela serve como base para tarefas como busca, análise de dados e execução de algoritmos complexos, influenciando a eficiência de acesso, modificação e processamento de dados.

O Merge Sort é um algoritmo de ordenação que consiste em dividir uma estrutura em subconjuntos e aplicar a ordenação nos elementos extraídos da estrutura original. Após a ordenação desses subconjuntos, é feita a mistura (merge) em um conjunto final ordenado. Podemos dizer literalmente que ele se utiliza daquela boa e velha frase que conhecemos: ”Dividir e conquistar”.

Este estudo visa não apenas explorar as propriedades e a implementação do Merge Sort, mas também comparar suas diferentes implementações em sete linguagens de programação distintas: C, C++, C#, Rust, Java, JavaScript, PHP e Python. Além disso, investigaremos generalizações do Merge Sort e analisaremos como diferentes abordagens de compilação e interpretação podem impactar seu desempenho, uma vez que baseado no número de comparações utilizadas, na utilização de recursão na implementação, na necessidade de memoria adicional e na abrangência do algoritimo, os seus resultados podem ser diferentes.

A. Contexto Histórico

Um dos desafios enfrentados pelos primeiros computadores era a ordenação eficiente de grandes volumes de dados. Ordenar dados é uma tarefa fundamental em muitos processos de computação, desde a organização de listas telefônicas até a execução de cálculos científicos. Na época (1945), os dados frequentemente não cabiam na memória limitada dos computadores e precisavam ser armazenados em dispositivos de armazenamento externo, como fitas magnéticas.

John von Neumann estava envolvido no desenvolvimento do EDVAC (Electronic Discrete Variable Automatic Computer), um dos primeiros computadores eletrônicos digitais. Ao contrário de seus predecessores, que usavam componentes eletromecânicos, o EDVAC usava válvulas termiônicas para realizar cálculos. Uma das contribuições fundamentais de von Neumann foi a arquitetura de von Neumann, que descrevia uma estrutura de computador com uma unidade de processamento central, memória, e dispositivos de entrada/saída. (Knuth, 1998) [1].

Os dispositivos de armazenamento externo tinham características diferentes da memória interna. Por exemplo, acessar dados em fitas magnéticas era sequencial, o que significava que encontrar um item específico podia ser demorado. Assim, era necessário desenvolver métodos de ordenação que minimizassem o número de acessos a esses dispositivos.

Von Neumann aplicou a estratégia ”divide e conquista” ao problema da ordenação. Ele percebeu que dividir o problema em subproblemas menores poderia facilitar a ordenação, especialmente quando lidando com grandes volumes de dados. O Merge Sort divide repetidamente o array em metades até que cada subarray contenha apenas um elemento (fase de divisão). Em seguida, os subarrays são combinados de forma ordenada (fase de conquista).

O Merge Sort foi particularmente útil para ordenação em fitas magnéticas, onde a leitura sequencial era mais eficiente do

que a leitura aleatória. A combinação ordenada dos subarrays (merge) podia ser feita eficientemente lendo dados sequencialmente de duas fitas de entrada e escrevendo os dados combinados em uma fita de saída.

A criação do Merge Sort por John von Neumann foi uma das primeiras demonstrações do poder da estratégia de ”divide e conquista” em algoritmos de ordenação. Seu trabalho pioneiro estabeleceu princípios que ainda são fundamentais na ciência da computação moderna. O Merge Sort continua a ser amplamente estudado e utilizado, tanto em implementações práticas quanto em contextos educacionais, como um exemplo clássico de algoritmo eficiente e elegante.

II. Metodologia

A. Descrição do Merge Sort

O Merge Sort é um algoritmo de ordenação baseado na técnica ”dividir e conquistar”: que ocorre o desmembramento do problema em vários subproblemas que são semelhantes ao problema original, mas de menor tamanho, resolvendo esses subproblemas recursivamente e, em seguida, combinando as soluções para criar uma solução para o problema original. Dessa forma ele opera, nos passos principais:

- **Divisão:** O array original é dividido recursivamente em subarrays menores até que cada subarray contenha apenas um elemento. Isso é feito dividindo repetidame’nte o array ao meio.
- **Conquista:** Os subarrays são ordenados recursivamente.
- **Combinação (Merge):** Os subarrays ordenados são mesclados para formar subarrays maiores, garantindo que os elementos mesclados estejam em ordem.

A recursão é ”finalizada” quando a sequência a ser ordenada atinge o comprimento 1, uma vez que, nessa situação, não há trabalho a ser realizado, uma vez que qualquer sequência com comprimento 1 já se encontra ordenada.

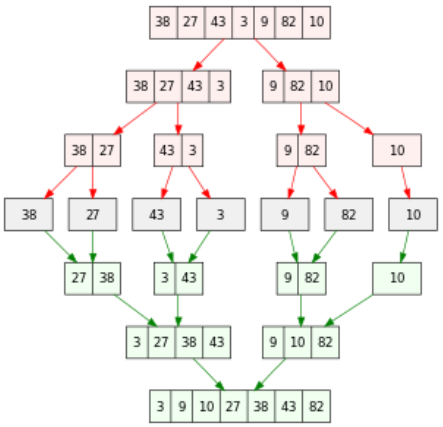

Fig. 1. Demonstração Merge Sort. Fonte: Wikipedia, 2024

Para entender melhor o funcionamento do Merge Sort, é importante analisar as duas funções principais que compõem o algoritmo: `Merge` e `MergeSort`.

A operação chave do algoritmo de ordenação é a intercalação de duas sequências ordenadas no passo de “combinação”. Para realizar a intercalação, utilizamos um procedimento auxiliar chamado `Merge(A, p, q, r)`, onde A é o array, p é o índice do primeiro elemento do primeiro subarray, q é o índice do último elemento do primeiro subarray e r é o índice do último elemento do segundo subarray, tais que $p \leq q < r$.

Esse procedimento assume que $A[p..q]$ (subarray esquerdo) e $A[q+1..r]$ (subarray direito) estão ordenados. Ele os mescla para formar um único subarray ordenado que substitui o subarray original $A[p..r]$. O procedimento `Merge` leva tempo $O(n)$ para mesclar dois subarrays de tamanho $n/2$ cada, onde $n = r - p + 1$ é o número total de elementos a serem mesclados. Em termos computacionais, cada passo básico do procedimento demanda um tempo constante, pois estamos comparando apenas os elementos superiores dos dois subarrays.

Para simplificar a implementação, utilizamos sentinelas assim como descrito por Cormen et al., 2012 [2], representadas por um valor infinito — valores especiais adicionados ao final de cada subarray para evitar verificações constantes de vazios. O valor sentinela ∞ garante que, ao ser exposto, ele não será escolhido como o menor elemento, a menos que todos os outros elementos já tenham sido processados. Veja o pseudocódigo a seguir:

Algorithm 1 Merge(A, p, q, r)

```
1:  n1 = q − p + 1
2:  n2 = r − q
3:  sejam L[1..n1 + 1] e R[1..n2 + 1] novos arranjos
4:  for i = 1 to n1 do
5:      L[i] = A[p + i − 1]
6:  end for
7:  for j = 1 to n2 do
8:      R[j] = A[q + j]
9:  end for
10: L[n1 + 1] = ∞
11: R[n2 + 1] = ∞
12: i = 1
13: j = 1
14: for k = p to r do
15:     if L[i] ≤ R[j] then
16:         A[k] = L[i]
17:         i = i + 1
18:     else
19:         A[k] = R[j]
20:         j = j + 1
21:     end if
22: end for
```

Detalhadamente, o procedimento Merge opera conforme a explicação a seguir:

1) *Primeira linha:* o comprimento n_1 do subconjunto $A[p..q]$ é calculado fazendo a subtração de q por p e adicionando 1.
2) *Segunda linha:* determina n_2, o comprimento do subconjunto $A[q+1..r]$, fazendo a subtração de r por q.

3) *Terceira linha:* Os arranjos L e R são criados com comprimentos $n_1 + 1$ e $n_2 + 1$, respectivamente, onde L se refere ao subarranjo esquerdo e R se refere ao subarranjo direito. Uma posição adicional é reservada em cada arranjo para conter uma sentinela.
4) *Quarta a sexta linha:* copia do subconjunto $A[p..q]$ para $L[1..n1]$, pois L é o subarranjo esquerdo.
5) *Sétima a nona linha:* copia de $A[q+1..r]$ para $R[1..n2]$, pois R é o subarranjo direito.
6) *Décima e décima primeira linha:* inserem sentinelas nas extremidades de L e R, garantindo que todos os elementos de A sejam copiados de volta para A.
7) *Décima segunda a vigésima segunda linha:* executam os $r - p + 1$ passos fundamentais, mantendo o invariante de loop da "Fig. 2".
   - No início das interações do loop `for` nas linhas 14 a 22, o subarranjo $A[p..k-1]$ contém os menores elementos de $L[1..n_1+1]$ e $R[1..n_2+1]$, em ordem.
   - $L[i]$ e $R[j]$ são os menores elementos em seus arranjos que ainda não foram copiados de volta para A
   - o invariante do loop é válido antes da primeira iteração do loop `for` nas linhas 14 a 22, que o invariante é mantido em cada iteração do loop e que o invariante fornece uma propriedade útil, usada para mostrar a correção no final do ciclo.

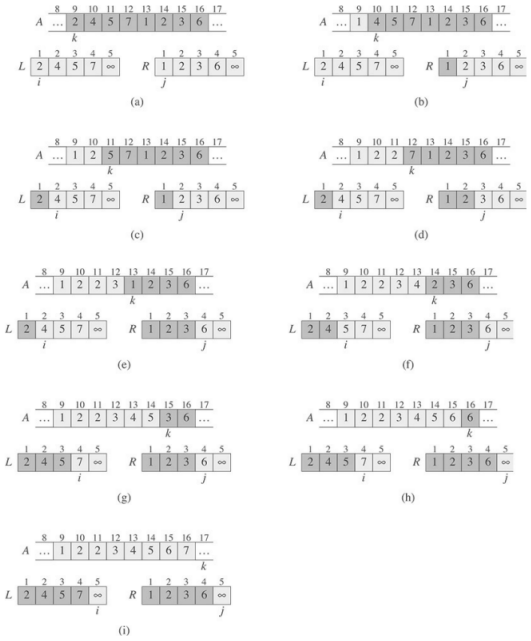

Fig. 2. Execução das linhas de 10 a 17 com a função Merge(A, 9, 12, 16). Cormen et al., 2012 [3].

Para ver que o processo de mesclagem é executado no tempo $\Theta(n)$, onde $n = r - p + 1$, pode observar que cada uma das linhas 1 a 3 e 10 a 13 requer tempo constante, ou seja, o loop `for` da linha 4 à linha 9 leva tempo $\Theta(n_1 + n_2) = \Theta(n)$, e o loop `for` da linha 14 à linha 22 tem n iterações, cada iteração levando tempo constante.

Agora pode-se usar o procedimento `Merge` como uma sub-rotina no algoritmo de ordenação por mesclagem. O procedimento `MergeSort(A, p, r)` ordena os elementos do subarranjo $A[p..r]$. Se $p \geq r$, o subarranjo tem no máximo um elemento e, portanto, já está ordenado. Caso contrário, a etapa de divisão simplesmente calcula um índice q que divide $A[p..r]$ em duas submatrizes: $A[p..q]$, contendo $n/2$ elementos, e $A[q+1..r]$, contendo $n/2$ elementos.

Algorithm 2 MergeSort

```
    function MERGESORT(A, p, r)
2:      if p < r then
            q ← ⌊(p + r)/2⌋
4:          MERGESORT(A, p, q)
            MERGESORT(A, q + 1, r)
6:          MERGE(A, p, q, r)
        end if
8:  end function
```

A função `MergeSort` então chama a si mesma recursivamente para classificar cada uma das duas submatrizes, e finalmente chama `Merge` para mesclar as duas submatrizes ordenadas.

B. Complexidade do Algoritmo

Concluindo então sua execução, o algoritmo de ordenação por mesclagem é executado em tempo $O(n \log n)$, onde n é o número de elementos a serem ordenados. A análise do tempo de execução do algoritmo de ordenação por mesclagem é baseada no fato de que a etapa de divisão leva tempo constante e que a etapa de mesclagem leva tempo $\Theta(n)$ para cada um dos subarranjos.

Quando um algoritmo contém uma chamada recursiva a si próprio, seu tempo de execução pode ser descrito na maioria das vezes por uma *equação de recorrência*, que descreve o tempo de execução global para um problema de tamanho n. Usando ferramentas matemáticas podemos resolver a recorrência e estabelecer limites para o desempenho do algoritmo.

O chamado *Teorema Mestre* é uma ferramenta útil para resolver recorrências de divisão e conquista. Ele fornece uma solução para recorrências da forma:

$$T(n) = aT(\frac{n}{b}) + f(n) \tag{1}$$

onde:

- $T(n)$ é o tempo de execução do algoritmo para uma entrada n.
- a é o número de subproblemas em que a instância original é dividida.
- $\frac{n}{b}$ é o tamanho de cada subproblema, onde b é um inteiro maior que 1.
- $f(n)$ é uma função assintoticamente positiva, que representa o custo de dividir o problema e combinar as soluções dos subproblemas.

No caso do Merge Sort, sabemos que $a = 2$, pois o algoritmo divide o array em dois subarrays de tamanho $n/2$ em cada chamada recursiva e que $\frac{n}{b} = \frac{n}{2}$. Como já dito anteriormente, este algoritmo leva um tempo $\Theta(n)$ para mesclar dois subarrays. Portanto, a função $f(n)$ é $\Theta(n)$. Assim, a recorrência do Merge Sort é dada por:

$$T(n) = 2T(\frac{n}{2}) + \Theta(n) \tag{2}$$

Cormem et al. (2012, p. 89) [4] descrevem que sendo $a \leq 1$ e $b > 1$ constantes, $f(n)$ uma função assintoticamente positiva e $T(n)$ uma função definida em inteiros não negativos pela recorrência $T(n) = aT(n/b) + f(n)$, então $T(n)$ tem a seguinte assíntota:

1) Caso 1: Se $f(n) = O(n^{\log_b a - \epsilon})$ para algum $\epsilon > 0$, então $T(n) = \Theta(n^{\log_b a})$.
2) Caso 2: Se $f(n) = \Theta(n^{\log_b a})$, então $T(n) = \Theta(n^{\log_b a} \log n)$.
3) Caso 3: Se $f(n) = \Omega(n^{\log_b a + \epsilon})$ para algum $\epsilon > 0$, e se $af(\frac{n}{b}) \leq kf(n)$ para algum $k < 1$ e n suficientemente grande, então $T(n) = \Theta(f(n))$.

Sabendo que o Merge Sort tem $f(n) = \Theta(n)$, e $log_b a = log_2 2 = 1$, podemos concluir que o caso 2 se aplica ao Merge Sort. Onde ficariamos com:

$$T(n) = \Theta(n^{\log_b a} \log n)$$
$$T(n) = \Theta(n^{\log_2 2} \log n)$$
$$T(n) = \Theta(n^1 \log n)$$
$$T(n) = \Theta(n \log n) \tag{3}$$

Portanto, através do Teorema Mestre, podemos concluir que a complexidade de tempo do Merge Sort é $\Theta(n \log n)$, onde n é o número de elementos a serem ordenados.

Além da complexidade de tempo, o Merge Sort também possui uma complexidade de espaço de $\mathcal{O}(n)$. Isso ocorre porque o algoritmo requer espaço adicional para armazenar os subarrays temporários durante o processo de divisão e combinação. Durante a fase de divisão, são criados subarrays temporários para armazenar as divisões recursivas do array original. Esses subarrays temporários são essenciais para garantir que o Merge Sort funcione corretamente, mas também adicionam uma sobrecarga de espaço proporcional ao tamanho do array de entrada.

C. Desempenho

O Merge Sort é considerado uns dos algoritmos de ordenação mais eficientes, principalmente por conta da sua estabilidade. Ele é considerado estável porque ele mantém a mesma complexidade de tempo em todos os casos. Para algoritmos de ordenação, no estudo da análise assintótica de tempo, é considerado que o melhor caso é quando o array já está ordenado, o pior caso é quando o array está ordenado de forma inversa e o caso médio é quando o array está ordenado de forma aleatória. No caso do Merge Sort, a complexidade de tempo é $\Theta(n \log n)$ para todos os casos, pois como ele divide o array em subarrays menores e os ordena independentemente, a ordem relativa dos elementos iguais é preservada, garantindo que a complexidade de tempo seja a mesma em todos os casos.

Já em termos de espaço, o Merge Sort requer espaço adicional para armazenar os subarrays temporários durante o processo de divisão e combinação. Durante a fase de divisão, são criados subarrays temporários para armazenar as divisões recursivas do array original. Esses subarrays temporários são essenciais para garantir que o Merge Sort funcione corretamente, mas também adicionam uma sobrecarga de espaço proporcional ao tamanho do array de entrada. Portanto, a complexidade de espaço do Merge Sort é $\mathcal{O}(n)$, onde n é o número de elementos a serem ordenados.

TABLE I
COMPLEXIDADE DO MERGE SORT.

Caso	Tempo	Espaço
Melhor	$n \log n$	n
Médio	$n \log n$	n
Pior	$n \log n$	n

D. Implementações

Para avaliar o desempenho do Merge Sort em diferentes linguagens de programação, implementamos o algoritmo em oito linguagens distintas: C, C++, C#, Rust, Java, JavaScript, PHP e Python. Cada implementação segue o mesmo princípio de divisão e conquista, mas difere em detalhes de sintaxe e estrutura de dados. Além da implementação do Merge Sort, também foi implementado para este um *manager* que realiza o gerenciamento de todos os códigos, permitindo a execução de todos os códigos de uma vez e a comparação de seus resultados, sabendo que para cada uma das linguagens, tipo de arquivo de entrada e quantidade de execuções, foi feita uma média de 10 execuções. Também, foi implementado um plotador de gráficos que permite a visualização dos resultados obtidos. Além disso, foi implementado algumas generalizações do Merge Sort, como o *K-Way Merge Sort*, *Custom Sequence Merge Sort* e *Field Based Merge Sort*, que são variações do Merge Sort que visam melhorar o desempenho do algoritmo em diferentes cenários. Todas as implementações estão disponíveis no repositório do GitHub: https://github.com/dudatsouza/Merge-Sort.

E. Dados de Entrada

Para avaliar o desempenho do Merge Sort, utilizamos uma série de conjuntos de dados de entrada com diferentes tipos de organização. Os conjuntos de dados de entrada são compostos por sequências de números inteiros gerados de forma aleatória, crescente, decrescente, quase crescente e quase decrescente, e t odos os arquivos possuem 1.000.000 de elementos, variando de 0 a 1.000.000.

Existe 3 arquivos de entrada de forma aleatória diferentes, gerados por bibliotecas da linguagem Python (Random, Numpy e Secrets), com a finalidade de garantir a aleatoriedade dos dados. A diferença entre as bibliotecas é que a biblioteca Random é a biblioteca padrão do Python gera números a partir

de um valor inicial chamado de semente, que é um valor inteiro que é passado para a função de geração de números aleatórios. A biblioteca Numpy é uma biblioteca de código aberto que fornece suporte para arrays e matrizes multidimensionais, além de funções matemáticas para operações com esses arrays. Já a biblioteca Secrets é uma biblioteca de código aberto que fornece funções para geração de números aleatórios seguros, que são números que são gerados de forma aleatória e imprevisível, o que é importante para garantir a segurança de sistemas de criptografia e autenticação.

Os arquivos com os dados ordenados de forma quase crescente e quase decrescente foram gerados a partir de um arquivo de dados ordenados de forma crescente, onde foi feita uma permutação aleatória de 10% dos dados.

F. Ambiente de Execução

Os experimentos foram realizados em um computador com as seguintes especificações:

- *Sistema Operacional:* Ubuntu 24.04 LTS - 64 bits
- *Modelo de hardware:* Dell Inspiron 13 5330
- *Processador:* Intel Core i7-1360P (18MB Cache, up to 5.00 GHz)
- *Memória RAM:* 16GB 4800MHz LPDDR5 Memory Onboard
- *Armazenamento:* 512GB M.2 PCIe NVMe Solid State Drive
- *Placa de Vídeo:* Intel(R) Iris(R) Xe Graphics

III. Modelos de Aplicação e Generalizações

A eficiência na ordenação de dados é uma preocupação central em ciência da computação e áreas afins. Entre os diversos algoritmos desenvolvidos para essa finalidade, o Merge Sort se destaca pela sua eficácia e simplicidade conceitual. Assim vamos explorar agora diferentes modelos de aplicação do Merge Sort, examinando suas vantagens e desvantagens, e adaptações em diferentes contextos computacionais.

A. Modelos de aplicação do Merge Sort

- *Grandes Volumes de Dados*: Para grandes volumes de dados o Merge costuma se destacar, pois é um algoritmo de fácil implementação, principalmente quando comparado a outros de ordenação. De forma mais técnica, uma vantagem que ele apresenta é sua complexidade de tempo a qual mostra que o método é mais eficiente e consistente quando comparado a outros, como algoritmos que tenham complexidade de tempo $O(n^2)$. Ele apresenta também para esse caso seu maior problema que é a grande utilização de memória, a qual compromete sistemas com restrição severas de memória. Embora seu desempenho geralmente compense a sobrecarga.
- *Dados Distribuídos*: Dados distribuídos são comuns em sistemas distribuídos, onde diferentes partes de um conjunto de dados estão localizadas em diferentes locais físicos. O Merge Sort é um método eficaz para ordenar dados distribuídos, pois sua abordagem de ”dividir e conquistar” permite que os dados sejam ordenados em pequenas partes, minimizando a transferência de grandes volumes de dados de uma vez. Além disso, o Merge Sort é um algoritmo estável, o que significa que a ordem relativa dos elementos iguais é mantida, o que é importante em sistemas complexos. Diferentes de outros algoritmos de ordenação, o Merge Sort pode ser aplicado a diferentes tipos de dados, enquanto são restritos a arrays.

B. Vantagens do Merge Sort

Todo algoritmo de ordenação tem suas vantagens e desvantagens, o que faz diferença na escolha do método a ser utilizado em diferentes contextos. Vamos explorar algumas das vantagens do Merge Sort em diferentes cenários:

- *Eficiência Assintótica*: Sua principal vantagem é o fato dele ter uma boa complexidade de tempo, $\Theta(n \log n)$, em relação a outros algoritmos de ordenação que normalmente possuem complexidade de tempo $O(n^2)$. Isso faz com que o Merge Sort seja uma escolha eficiente para ordenar grandes volumes de dados, pois seu desempenho é mais consistente e previsível em diferentes cenários.
- *Estabilidade*: Como já foi citado antes no artigo, o método Merge Sort é estável, o que faz com que em diferentes casos, aleatórios ou não, o seu tempo de execução seja o mesmo. Isso é uma vantagem importante, pois garante que a ordem relativa dos elementos iguais seja mantida, o que é crucial em muitos contextos computacionais.
- *Adaptação a Diferentes Estruturas de Dados*: Uma grande vantagem do Merge Sort é o fato dele não ser restrito apenas a arrays, assim sendo ele pode ordenar listas encadeadas, árvores, e outros tipos de estruturas de dados. Ele consegue fazer isso, por conta de sua abordagem recursiva e à manipulação de ponteiros ou referências, em vez de depender de acesso aleatório como alguns outros algoritmos de ordenação.
- *Facilidade de Implementação e Manutenção*: Quando comparado a outros métodos de ordenação, como o Quick Sort ou o Heap Sort, o Merge Sort se destaca pela sua facilidade. Dessa forma, é possível uma melhor averiguação e manutenção do código, contribuindo para uma implementação robusta e confiável.

Posto todas essas vantagens do método Merge Sort, nota-se que apesar de simples entendimento é um algoritmo capaz de tratar de diversos casos, possuindo vantagens muito importantes quando pensadas dentro de aplicações práticas, em ambientes de produtividade.

C. Desvantagens do Merge Sort

Como qualquer algoritmo, o Merge Sort também possui desvantagens que podem afetar seu desempenho em determinados cenários. Vamos explorar algumas das desvantagens do Merge Sort e discutir como elas podem impactar sua eficácia em diferentes contextos:

- *Uso de Memória*: Quando se trata de memoria o Merge Sort apresenta algumas desvantagens, tendo em vista que não é in-place. Primeiramente, o método necessita de uma

memoria adicional O(n), por conta que ele divide o array em sub-arrays e utiliza arrays auxiliares para mesclá-los. Necessitando dessa forma do ”dobro” de memoria. Assim, em sistemas com memória limitada, isso pode ser um problema significativo resultando em perca de desempenho. Além disso, o método acessa diversas vezes a memoria cache da CPU de forma aleatória, uma vez que durante a divisão dos arrays eles podem ser salvos em lugares distantes da memoria. Diferente de algoritmos como o Quick Sort que, geralmente, acessa a memória de maneira mais local. Assim o Merge Sort pode ser menos eficiente em termos de utilização da memória cache, ocasionando até em alguns casos perca de desempenho.

- *Desempenho em certos casos*: O Merge Sort apesar de ser fácil de ser aplicado, ele é um algoritmo normalmente utilizado para fins mais complexos. Dessa forma, para arrays pequenos ou já quase ordenados, algoritmos mais simples, como o Insertion Sort, podem ser mais eficientes. O Merge Sort pode ser excessivamente complexo para essas situações, tanto em termos de tempo quanto de uso de memória. Assim também é importante notar que ao depender do caso, outros algoritmos podem apresentar maneira mais fácil de realizar a ordenação, tendo menos chance de erros.

D. Exemplos Práticos

- **Ordenação de Grandes Arquivos de Dados**: Utilizado em sistemas de banco de dados para ordenar grandes volumes de dados que não cabem na memória principal.
- **Integração de Dados**: Durante a integração de dados de diferentes fontes, é comum precisar ordenar os dados para eliminar duplicatas, mesclar registros ou preparar os dados para análises subsequentes. O Merge Sort é frequentemente usado para garantir que esses dados sejam ordenados eficientemente.
- **Aplicações em Bioinformática**: Ordenação de sequências de DNA onde a estabilidade da ordenação é crucial para manter a integridade dos dados.
- **Aplicações Web**: Utilização do Merge Sort em scripts de servidor para manipulação e ordenação de dados recebidos do cliente.
- **Redes de Telecomunicações**: Em redes de telecomunicações, os pacotes de dados podem precisar ser ordenados por tempo de chegada ou outros critérios para garantir a entrega correta e eficiente. O Merge Sort pode ser usado para ordenar esses pacotes de forma eficiente.

E. Generalizações

As generalizações do Merge Sort são adaptações do algoritmo original para atender a requisitos específicos de diferentes aplicações. Essas generalizações podem incluir modificações para lidar com diferentes tipos de dados, otimizar o desempenho em cenários específicos ou estender a funcionalidade do algoritmo básico. Vamos explorar algumas generalizações comuns do Merge Sort e discutir suas implementações detalhadas e resultados.

1) Generalizações Implementadas: Foram feitas as implementações de 3 generalizações do Merge Sort e vamos citar também outras generalizações que podem ser feitas, mas que não foram implementadas neste trabalho.

a) Tipos de Dados Diferentes (Custom Sequence): Adaptação do Merge Sort para ordenar tipos de dados além de números, como strings, objetos complexos ou outras estruturas de dados. Por exemplo, o algoritmo pode ser generalizado para ordenar uma lista de objetos com base em um atributo específico. A capacidade de fornecer uma função de comparação personalizada permite que o algoritmo seja adaptado para atender a requisitos específicos de ordenação. Vemos um exemplo de aplicação prática dessa generalização em sistemas de gerenciamento de banco de dados, onde é necessário ordenar registros com base em diferentes critérios.

Para a implementação usamos uma sequência de caracteres específicos e passamos eles como parâmetros para a função de ordenação. A função de comparação é adaptada para lidar com diferentes tipos de dados. A função de comparação personalizada é passada como um parâmetro para a função de mesclagem, permitindo que o Merge Sort ordene os objetos com base em um campo específico.

b) Usando Outros Campos de Dados (Field Based): Adaptação do Merge Sort para ordenar uma lista de objetos com base em um campo específico. Por exemplo, o algoritmo pode ser generalizado para ordenar uma lista de objetos com base em um atributo específico, como um campo de data, um campo numérico ou um campo de texto. Essa generalização é útil em cenários onde a ordenação é necessária com base em critérios específicos.

A implementação utiliza uma função de comparação personalizada que compara os objetos com base em um campo específico. A função de comparação é passada como um parâmetro para a função de mesclagem, permitindo que o Merge Sort ordene os objetos com base em um campo específico. A implementação é semelhante à generalização de tipos de dados diferentes, mas com foco em campos específicos de objetos.

c) Ordenação em K-Caminhos (K-Way Merge Sort): Adaptação do Merge Sort para dividir o array em k partes e mesclar as partes de maneira eficiente. Essa abordagem é útil quando o array é muito grande para ser ordenado em memória e precisa ser dividido em partes menores para processamento paralelo ou distribuído. A ordenação em k-caminhos é uma generalização do Merge Sort que divide o array em k partes e mescla as partes de maneira eficiente.

A implementação divide o array em k partes e ordena cada parte separadamente. Em seguida, as partes ordenadas são mescladas em um único array ordenado. A implementação pode ser feita de forma recursiva ou iterativa, dependendo dos requisitos de desempenho e memória. O que muda em relação ao Merge Sort tradicional na sua função `Merge` é que ela recebe um parâmetro adicional, que é o número de caminhos a serem mesclados. E na função `MergeSort` é feita a divisão

do array em k partes e a chamada recursiva do Merge Sort para cada parte.

2) Outras Generalizações:

a) Parallel Merge Sort: O processo de paralelização do algoritmo Merge Sort envolve dividir o trabalho de ordenação entre múltiplos processadores ou núcleos de CPU para aumentar a eficiência e reduzir o tempo de execução. Aqui estão os passos principais envolvidos:

- *Divisão do Array:* O array inicial é dividido em subarrays menores, geralmente de tamanho igual ou aproximadamente igual, dependendo do número de processadores disponíveis.
- *Ordenação Paralela:* Cada subarray é então ordenado de forma independente por diferentes threads ou processos simultaneamente. Cada thread/processo pode executar o algoritmo Merge Sort no seu próprio subarray.
- *Mesclagem Paralela:* Após a ordenação de cada subarray, os resultados parciais são mesclados em pares. Isso pode ser feito de maneira recursiva ou iterativa, aproveitando os diferentes processadores para realizar as operações de mesclagem simultaneamente.
- *Combinação dos Resultados:* Conforme os subarrays são mesclados, os resultados são combinados até que o array inicial esteja completamente ordenado.

A paralelização pode oferecer uma melhoria significativa no desempenho do Merge Sort, especialmente em sistemas com múltiplos núcleos de processamento. No entanto, é importante considerar a sobrecarga de comunicação entre os processadores, a sincronização necessária entre threads/processos e a distribuição equilibrada do trabalho para obter os melhores resultados de desempenho.

Implementar o Merge Sort de forma paralela pode ser feito utilizando técnicas como programação concorrente em linguagens que suportam threads (como Java, C++ com threads padrão ou bibliotecas como pthreads em C) ou utilizando modelos de programação paralela como MPI (Message Passing Interface) em ambientes distribuídos. Essa implementação envolve alguns passos específicos para dividir o trabalho entre múltiplos threads ou processos. Veja os passos principais:

- *Divisão do Trabalho:* Primeiro, você precisa dividir o array em subarrays menores para que cada thread possa ordenar independentemente seu próprio subarray.
  - *Divisão Inicial:* Divida o array em partes iguais ou aproximadamente iguais, dependendo do número de threads disponíveis e do tamanho do array.
- *Criação de Threads:* Crie um conjunto de threads, cada um responsável por ordenar um subarray específico.
  - *Início dos Threads:* Cada thread iniciará a ordenação do seu subarray usando o algoritmo Merge Sort.
- *Ordenação Paralela:* Cada thread ordena seu subarray de maneira independente. Precisa de uma função principal que divide o trabalho entre duas threads para ordenar recursivamente as duas metades do array. Você pode adaptar o número de threads conforme necessário, gerenciando o balanceamento entre o número de threads e o tamanho do array para obter o melhor desempenho.
- *Mesclagem Paralela:* Depois que os threads terminarem de ordenar seus respectivos subarrays, a mesclagem dos subarrays ordenados também pode ser feita de forma paralela. Isso pode ser implementado usando técnicas de sincronização e divisão do trabalho de mesclagem entre os threads disponíveis.

Implementar o Merge Sort de forma paralela requer um bom entendimento de programação concorrente e como distribuir o trabalho entre os threads de maneira eficiente. Além disso, é importante gerenciar a sincronização e a comunicação entre os threads para evitar condições de corrida e garantir que a ordenação paralela seja correta e eficiente.

b) Ordenação Externa: A ordenação externa do Merge Sort é utilizada quando os dados a serem ordenados não cabem todos na memória principal (RAM). Nesse caso, os dados são geralmente armazenados em um dispositivo de armazenamento secundário, como um disco rígido, e são lidos em blocos menores que podem ser manipulados na memória principal. Veja:

- *Divisão em Blocos:* Os dados são lidos do armazenamento secundário em blocos que cabem na memória principal. A quantidade de dados lida de cada vez depende da capacidade da memória RAM disponível.
- *Ordenação Interna:* Cada bloco de dados é ordenado internamente na memória principal usando o Merge Sort. Isso envolve dividir recursivamente cada bloco até que cada subbloco possa ser ordenado individualmente.
- *Fusão dos Blocos:* Após a ordenação interna de cada bloco, os blocos ordenados são fundidos (merged) em um único arquivo de saída ordenado. Esse processo de fusão é semelhante ao passo de fusão do Merge Sort convencional, mas agora é aplicado para combinar blocos de dados ao invés de subarrays.
- *Escrita no Disco:* O arquivo de saída final, que contém todos os dados ordenados, é escrito de volta no armazenamento secundário.

Durante todo esse processo, é crucial gerenciar eficientemente a leitura e a escrita de blocos de dados no armazenamento secundário para minimizar o número de operações de leitura/escrita, pois essas operações tendem a ser mais lentas em comparação com o processamento na memória principal.

A implementação da Ordenação Externa do Merge Sort (também conhecida como Merge Sort Externo) pode variar dependendo da linguagem e das bibliotecas específicas utilizadas, mas vou descrever um exemplo básico de como isso poderia ser implementado de maneira geral, usando conceitos que são comuns em muitas implementações.

- *Leitura de Blocos Iniciais:* Os dados são lidos do arquivo ou dispositivo de armazenamento secundário em blocos que cabem na memória principal. Isso pode ser feito usando operações de leitura de arquivo.
- *Ordenação Interna dos Blocos:* Cada bloco de dados lido é ordenado internamente na memória principal usando o

algoritmo Merge Sort. Este passo é similar à ordenação convencional do Merge Sort, mas aplicada a blocos menores de dados.

- *Fusão dos Blocos Ordenados:* Após a ordenação interna de cada bloco, os blocos são fundidos (merged) para formar blocos maiores, que também são ordenados. Esse processo de fusão é semelhante ao passo de fusão do Merge Sort convencional, mas agora aplicado a blocos de dados em vez de subarrays.
- *Escrita dos Blocos Ordenados:* Os blocos maiores, que foram ordenados após a fusão dos blocos menores, são escritos de volta no armazenamento secundário. Isso pode ser feito usando operações de escrita em arquivo.
- *Fusão Final:* Quando todos os blocos foram ordenados e escritos, é realizada uma última fusão para combinar todos os blocos ordenados em um único arquivo ou conjunto de dados ordenados final. Isso é feito usando o processo de fusão do Merge Sort tradicional.

É importante lembrar que a eficiência da ordenação externa depende do tamanho dos blocos de dados que são lidos e escritos. Um tamanho de bloco ideal deve ser determinado para equilibrar entre o número de operações de leitura/escrita e a capacidade de ordenação eficiente na memória principal. Também o gerenciamento adequada da memória é essencial para garantir que os blocos de dados sejam manipulados de maneira eficiente na memória principal, especialmente se a memória disponível for limitada. Além de que a performance da ordenação externa é fortemente influenciada pela eficiência das operações de leitura e escrita no armazenamento secundário. Minimizar o número de acessos ao disco e maximizar a eficiência da ordenação interna são chave para um bom desempenho.

c) Distribuição em Redes: O Merge Sort pode ser distribuído em redes de computadores para processamento paralelo em ambientes distribuídos. Nesse cenário, os dados a serem ordenados são divididos em subarrays menores e distribuídos entre os nós da rede. Cada nó executa o algoritmo Merge Sort em seu subarray local e, em seguida, os resultados parciais são mesclados em pares por meio de comunicação entre os nós. Isso é aplicado quando o conjunto de dados é muito grande para ser processado em um único computador ou quando a velocidade de processamento é crítica, como em sistemas de análise de big data.

d) Estruturas de Dados Diferentes: O Merge Sort pode ser adaptado para ordenar diferentes tipos de estruturas de dados, como listas encadeadas, árvores binárias, grafos e outros tipos de dados complexos. A adaptação do algoritmo para lidar com diferentes estruturas de dados envolve a definição de funções de comparação personalizadas e a implementação de operações de mesclagem específicas para cada tipo de dado.

e) Mais Algumas: Além das mencionadas, existem outras generalizações como o uso do Merge Sort em algoritmos de compressão de dados, em sistemas de arquivos e em diversas outras aplicações de ciência da computação e engenharia de software.

IV. Resultados e Discussões

A. Apresentação dos Resultados

Os resultados obtidos nos experimentos realizados com o Merge Sort em oito linguagens de programação diferentes onde os tempos de execução são medidos em segundos e representam a média de 10 execuções para cada conjunto de dados de entrada. Como todos os arquivos de entradas possuem 1.000.000 de elementos, para fazer a análisse em relação ao tamanho da entrada, foram feitos testes onde cada algoritmo pegava até a quantidade de elementos definida. Os resultados são apresentados em tabelas e gráficos para facilitar a visualização e comparação dos dados.Além disso, foram feita dois tipos de coletas de dados, uma analisando o desempenho de acordo com os arquivos de entrada e outro analisando o desempenho de acordo com cada linguagem de programação.

1) Desempenho de Acordo com os Arquivos de Entrada:

a) Dados de Entrada Aleatórios - random1.txt: A tabela II e o gráfico da figura 3 apresentam os tempos de execução do Merge Sort em segundos para dados de entrada aleatórios gerados pela biblioteca Random da linguagem Python.

TABLE II
Tempos de Execução do Merge Sort com Dados de Entrada Aleatórios - Biblioteca Random

Linguagem	100	1000	10000	100000	500000	1000000
C	$1.94x10^{-5}$	$2.56x10^{-4}$	$2.92x10^{-3}$	$3.74x10^{-2}$	$1.57x10^{-1}$	$3.32x10^{-1}$
C++	$3.17x10^{-5}$	$3.38x10^{-4}$	$3.95x10^{-3}$	$4.53x10^{-2}$	$2.37x10^{-1}$	$4.89x10^{-1}$
C#	$3.75x10^{-2}$	$3.83x10^{-2}$	$4.29x10^{-2}$	$1.11x10^{-1}$	$3.41x10^{-1}$	$6.28x10^{-1}$
Rust	$1.57x10^{-5}$	$1.31x10^{-4}$	$1.55x10^{-3}$	$1.80x10^{-2}$	$9.67x10^{-2}$	$1.82x10^{-1}$
Java	$1.49x10^{-3}$	$6.48x10^{-3}$	$1.76x10^{-2}$	$1.57x10^{-1}$	$4.54x10^{-1}$	$8.56x10^{-1}$
JavaScript	$4.38x10^{-4}$	$2.50x10^{-3}$	$4.55x10^{-2}$	$5.00x10^{-1}$	$2.08x10^{-1}$	$4.27x10^{-1}$
PHP	$8.31x10^{-5}$	$9.05x10^{-4}$	$1.15x10^{-2}$	$1.29x10^{-1}$	$7.06x10^{-1}$	$1.49x10^{-1}$
Python	$1.39x10^{-4}$	$1.67x10^{-3}$	$2.12x10^{-2}$	$2.37x10^{-1}$	$1.38x10^{-1}$	$3.06x10^{-1}$

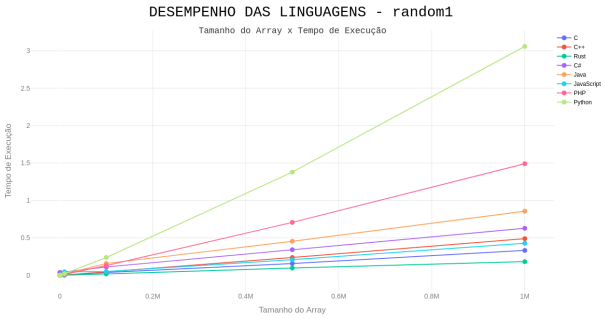

Fig. 3. Tempos de Execução do Merge Sort com Dados de Entrada Aleatórios - Biblioteca Random

b) Dados de Entrada Aleatórios - random2.txt: A tabela III e o gráfico da figura 4 apresentam os tempos de execução do Merge Sort em segundos para dados de entrada aleatórios gerados pela biblioteca Numpy da linguagem Python.

TABLE III
TEMPOS DE EXECUÇÃO DO MERGE SORT COM DADOS DE ENTRADA ALEATÓRIOS - BIBLIOTECA NUMPY

Linguagem	100	1000	10000	100000	500000	1000000
C	$1.91x10^{-5}$	$2.42x10^{-4}$	$2.93x10^{-3}$	$3.55x10^{-2}$	$1.58x10^{-1}$	$3.35x10^{-1}$
C++	$3.18x10^{-5}$	$3.87x10^{-4}$	$4.07x10^{-3}$	$4.54x10^{-2}$	$2.38x10^{-1}$	$4.95x10^{-1}$
C#	$3.74x10^{-2}$	$3.75x10^{-2}$	$4.28x10^{-2}$	$1.11x10^{-1}$	$3.41x10^{-1}$	$6.28x10^{-1}$
Rust	$1.83x10^{-5}$	$1.55x10^{-4}$	$1.67x10^{-3}$	$1.85x10^{-2}$	$9.42x10^{-2}$	$1.83x10^{-1}$
Java	$1.38x10^{-3}$	$6.48x10^{-3}$	$1.54x10^{-2}$	$1.57x10^{-1}$	$4.54x10^{-1}$	$8.56x10^{-1}$
JavaScript	$4.38x10^{-4}$	$2.50x10^{-3}$	$4.55x10^{-2}$	$5.00x10^{-1}$	$2.08x10^{-1}$	$4.27x10^{-1}$
PHP	$8.31x10^{-5}$	$9.05x10^{-4}$	$1.15x10^{-2}$	$1.29x10^{-1}$	$7.06x10^{-1}$	$1.49x10^{-1}$
Python	$1.39x10^{-4}$	$1.67x10^{-3}$	$2.12x10^{-2}$	$2.37x10^{-1}$	$1.38x10^{-1}$	$3.06x10^{-1}$

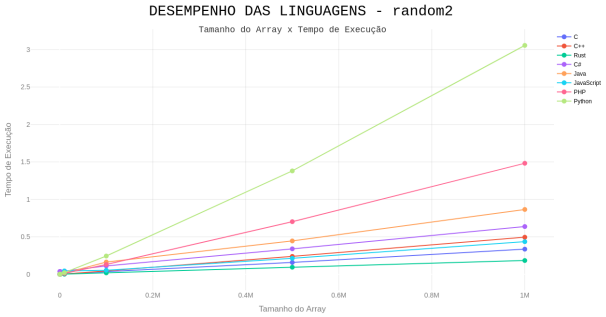

Fig. 4. Tempos de Execução do Merge Sort com Dados de Entrada Aleatórios - Biblioteca Numpy

c) Dados de Entrada Aleatórios - random3.txt: A tabela IV e o gráfico da figura 5 apresentam os tempos de execução do Merge Sort em segundos para dados de entrada aleatórios gerados pela biblioteca Secrets da linguagem Python.

TABLE IV
TEMPOS DE EXECUÇÃO DO MERGE SORT COM DADOS DE ENTRADA ALEATÓRIOS - BIBLIOTECA SECRETS

Linguagem	100	1000	10000	100000	500000	1000000
C	$1.95x10^{-5}$	$2.85x10^{-4}$	$3.48x10^{-3}$	$3.48x10^{-2}$	$1.70x10^{-1}$	$3.36x10^{-1}$
C++	$3.10x10^{-5}$	$3.93x10^{-4}$	$4.07x10^{-3}$	$4.39x10^{-2}$	$2.48x10^{-1}$	$4.99x10^{-1}$
C#	$3.75x10^{-2}$	$3.83x10^{-2}$	$4.29x10^{-2}$	$1.11x10^{-1}$	$3.41x10^{-1}$	$6.28x10^{-1}$
Rust	$1.85x10^{-5}$	$1.67x10^{-4}$	$1.67x10^{-3}$	$1.85x10^{-2}$	$9.73x10^{-2}$	$1.83x10^{-1}$
Java	$1.49x10^{-3}$	$6.48x10^{-3}$	$1.57x10^{-2}$	$1.57x10^{-1}$	$4.54x10^{-1}$	$8.56x10^{-1}$
JavaScript	$4.38x10^{-4}$	$2.50x10^{-3}$	$4.55x10^{-2}$	$5.00x10^{-1}$	$2.08x10^{-1}$	$4.27x10^{-1}$
PHP	$8.31x10^{-5}$	$9.05x10^{-4}$	$1.15x10^{-2}$	$1.29x10^{-1}$	$7.06x10^{-1}$	$1.49x10^{-1}$
Python	$1.39x10^{-4}$	$1.67x10^{-3}$	$2.12x10^{-2}$	$2.37x10^{-1}$	$1.38x10^{-1}$	$3.06x10^{-1}$

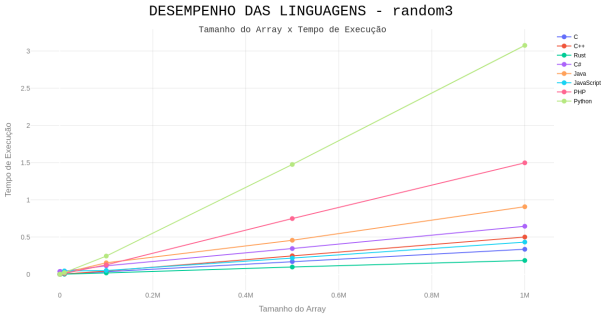

Fig. 5. Tempos de Execução do Merge Sort com Dados de Entrada Aleatórios - Biblioteca Secrets

d) Dados de Entrada Crescentes - ascending.txt: A tabela V e o gráfico da figura 6 apresentam os tempos de execução do Merge Sort em segundos para dados de entrada ordenados de forma crescente.

TABLE V
TEMPOS DE EXECUÇÃO DO MERGE SORT COM DADOS DE ENTRADA CRESCENTES

Linguagem	100	1000	10000	100000	500000	1000000
C	$1.34x10^{-5}$	$1.49x10^{-4}$	$1.62x10^{-3}$	$1.97x10^{-2}$	$8.67x10^{-2}$	$1.72x10^{-1}$
C++	$2.50x10^{-5}$	$2.55x10^{-4}$	$2.56x10^{-3}$	$3.06x10^{-2}$	$1.47x10^{-1}$	$3.01x10^{-1}$
C#	$3.71x10^{-2}$	$3.83x10^{-2}$	$4.29x10^{-2}$	$1.11x10^{-1}$	$3.41x10^{-1}$	$6.28x10^{-1}$
Rust	$1.61x10^{-5}$	$1.83x10^{-4}$	$1.67x10^{-3}$	$1.85x10^{-2}$	$9.73x10^{-2}$	$1.83x10^{-1}$
Java	$1.49x10^{-3}$	$6.48x10^{-3}$	$1.40x10^{-2}$	$1.57x10^{-1}$	$4.54x10^{-1}$	$8.56x10^{-1}$
JavaScript	$4.19x10^{-4}$	$2.50x10^{-3}$	$4.47x10^{-2}$	$5.00x10^{-1}$	$2.08x10^{-1}$	$4.27x10^{-1}$
PHP	$7.75x10^{-5}$	$7.51x10^{-4}$	$9.14x10^{-3}$	$1.29x10^{-1}$	$5.46x10^{-1}$	$1.12x10^{-1}$
Python	$1.33x10^{-4}$	$1.51x10^{-3}$	$1.80x10^{-2}$	$2.23x10^{-1}$	$1.06x10^{-1}$	$2.23x10^{-1}$

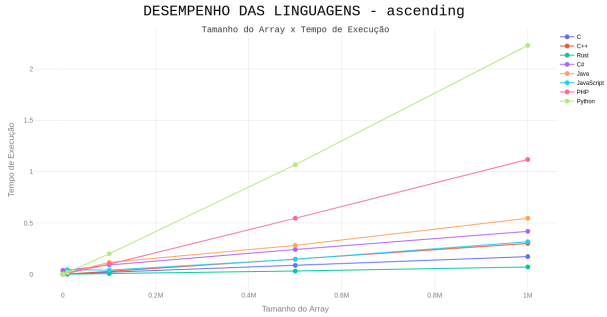

Fig. 6. Tempos de Execução do Merge Sort com Dados de Entrada Crescentes

e) Dados de Entrada Decrescentes - descending.txt: A tabela VI e o gráfico da figura 7 apresentam os tempos de execução do Merge Sort em segundos para dados de entrada ordenados de forma decrescente.

TABLE VI
TEMPOS DE EXECUÇÃO DO MERGE SORT COM DADOS DE ENTRADA DECRESCENTES

Linguagem	100	1000	10000	100000	500000	1000000
C	$1.34x10^{-5}$	$2.50x10^{-5}$	$1.61x10^{-4}$	$2.11x10^{-3}$	$8.45x10^{-3}$	$1.70x10^{-2}$
C++	$2.36x10^{-5}$	$2.57x10^{-5}$	$2.57x10^{-4}$	$3.06x10^{-3}$	$1.47x10^{-2}$	$3.01x10^{-2}$
C#	$3.71x10^{-2}$	$3.83x10^{-2}$	$4.29x10^{-2}$	$1.11x10^{-1}$	$3.41x10^{-1}$	$6.28x10^{-1}$
Rust	$1.61x10^{-5}$	$1.83x10^{-4}$	$1.67x10^{-3}$	$1.85x10^{-2}$	$9.73x10^{-2}$	$1.83x10^{-1}$
Java	$1.49x10^{-3}$	$6.48x10^{-3}$	$1.71x10^{-2}$	$1.57x10^{-1}$	$4.54x10^{-1}$	$8.56x10^{-1}$
JavaScript	$4.19x10^{-4}$	$2.50x10^{-3}$	$4.47x10^{-2}$	$5.00x10^{-1}$	$2.08x10^{-1}$	$4.27x10^{-1}$
PHP	$7.75x10^{-5}$	$7.51x10^{-4}$	$9.14x10^{-3}$	$1.29x10^{-1}$	$5.46x10^{-1}$	$1.12x10^{-1}$
Python	$1.33x10^{-4}$	$1.51x10^{-3}$	$1.80x10^{-2}$	$2.23x10^{-1}$	$1.06x10^{-1}$	$2.23x10^{-1}$

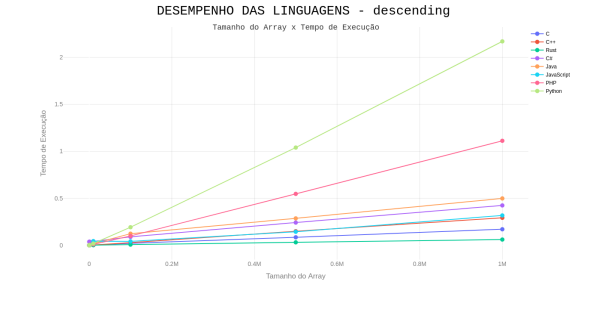

Fig. 7. Tempos de Execução do Merge Sort com Dados de Entrada Decrescentes

f) Dados de Entrada Quase Crescentes - nearly_sorted_ascending.txt: A tabela VII e o gráfico da figura 8 apresentam os tempos de execução do Merge Sort em segundos para dados de entrada ordenados de forma quase crescente.

TABLE VII
TEMPOS DE EXECUÇÃO DO MERGE SORT COM DADOS DE ENTRADA QUASE CRESCENTES

Linguagem	100	1000	10000	100000	500000	1000000
C	$1.34x10^{-5}$	$1.49x10^{-4}$	$1.62x10^{-3}$	$1.97x10^{-2}$	$8.67x10^{-2}$	$1.72x10^{-1}$
C++	$2.50x10^{-5}$	$2.55x10^{-4}$	$2.56x10^{-3}$	$3.06x10^{-2}$	$1.47x10^{-1}$	$3.01x10^{-1}$
C#	$3.71x10^{-2}$	$3.83x10^{-2}$	$4.29x10^{-2}$	$1.11x10^{-1}$	$3.41x10^{-1}$	$6.28x10^{-1}$
Rust	$1.61x10^{-5}$	$1.83x10^{-4}$	$1.67x10^{-3}$	$1.85x10^{-2}$	$9.73x10^{-2}$	$1.83x10^{-1}$
Java	$1.49x10^{-3}$	$6.48x10^{-3}$	$1.40x10^{-2}$	$1.57x10^{-1}$	$4.54x10^{-1}$	$8.56x10^{-1}$
JavaScript	$4.19x10^{-4}$	$2.50x10^{-3}$	$4.47x10^{-2}$	$5.00x10^{-1}$	$2.08x10^{-1}$	$4.27x10^{-1}$
PHP	$7.75x10^{-5}$	$7.51x10^{-4}$	$9.14x10^{-3}$	$1.29x10^{-1}$	$5.46x10^{-1}$	$1.12x10^{-1}$
Python	$1.33x10^{-4}$	$1.51x10^{-3}$	$1.80x10^{-2}$	$2.23x10^{-1}$	$1.06x10^{-1}$	$2.23x10^{-1}$

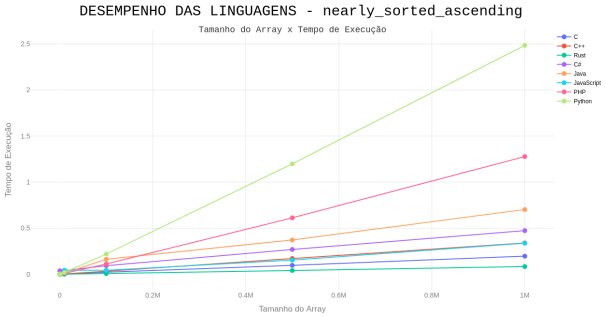

Fig. 8. Tempos de Execução do Merge Sort com Dados de Entrada Quase Crescentes

g) Dados de Entrada Quase Decrescentes - nearly_sorted_descending.txt: A tabela VIII e o gráfico da figura 9 apresentam os tempos de execução do Merge Sort em segundos para dados de entrada ordenados de forma quase decrescente.

TABLE VIII
TEMPOS DE EXECUÇÃO DO MERGE SORT COM DADOS DE ENTRADA QUASE DECRESCENTES

Linguagem	100	1000	10000	100000	500000	1000000
C	$1.34x10^{-5}$	$2.50x10^{-5}$	$1.61x10^{-4}$	$2.11x10^{-3}$	$8.45x10^{-3}$	$1.70x10^{-2}$
C++	$2.36x10^{-5}$	$2.57x10^{-5}$	$2.57x10^{-4}$	$3.06x10^{-3}$	$1.47x10^{-2}$	$3.01x10^{-2}$
C#	$3.71x10^{-2}$	$3.83x10^{-2}$	$4.29x10^{-2}$	$1.11x10^{-1}$	$3.41x10^{-1}$	$6.28x10^{-1}$
Rust	$1.61x10^{-5}$	$1.83x10^{-4}$	$1.67x10^{-3}$	$1.85x10^{-2}$	$9.73x10^{-2}$	$1.83x10^{-1}$
Java	$1.49x10^{-3}$	$6.48x10^{-3}$	$1.71x10^{-2}$	$1.57x10^{-1}$	$4.54x10^{-1}$	$8.56x10^{-1}$
JavaScript	$4.19x10^{-4}$	$2.50x10^{-3}$	$4.47x10^{-2}$	$5.00x10^{-1}$	$2.08x10^{-1}$	$4.27x10^{-1}$
PHP	$7.75x10^{-5}$	$7.51x10^{-4}$	$9.14x10^{-3}$	$1.29x10^{-1}$	$5.46x10^{-1}$	$1.12x10^{-1}$
Python	$1.33x10^{-4}$	$1.51x10^{-3}$	$1.80x10^{-2}$	$2.23x10^{-1}$	$1.06x10^{-1}$	$2.23x10^{-1}$

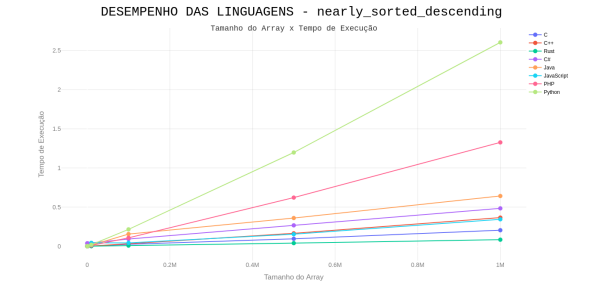

Fig. 9. Tempos de Execução do Merge Sort com Dados de Entrada Quase Decrescentes

2) Desempenho de Acordo com as Linguagens:

a) C: O gráfico a seguir apresenta o desempenho do Merge Sort em C para diferentes arquivos de entrada

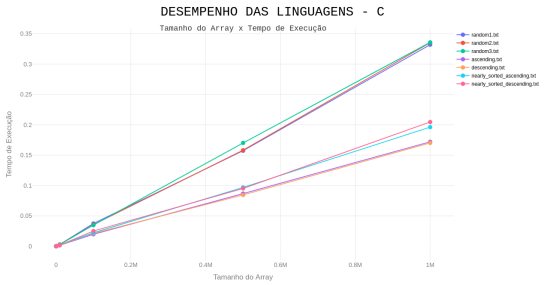

Fig. 10. Tempos de Execução do Merge Sort em C

Observe que mesmo sendo pouca a diferença entre os tipos de arquivos de entrada, mostrando que para os aleatórios o tempo de execução foi um pouco maior, porém ele se distancia um pouco de uma curva logarítmica, o que mostra que o algoritmo pode não está se comportando de forma esperada.

b) C++: Agora na figura 11 da linguagem C++.

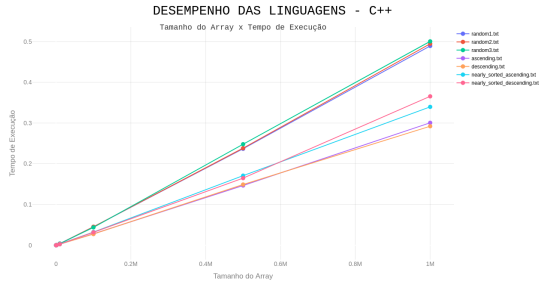

Fig. 11. Tempos de Execução do Merge Sort em C++

Observe aqui que o mesmo se repitiu, o tempo de execução para os arquivos aleatórios foi um pouco maior, porém ele também se distancia um pouco de uma curva logarítmica, o que mostra que o algoritmo pode não está se comportando de forma esperada.

c) C#: Aqui da linguagem C#.

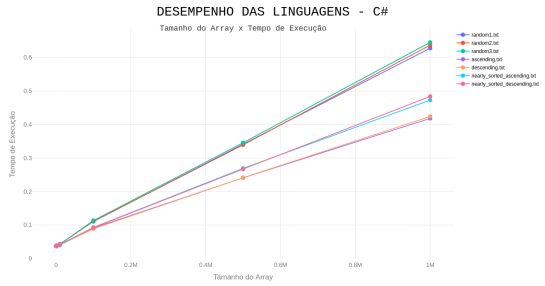

Fig. 12. Tempos de Execução do Merge Sort em C#

Aqui se repete o mesmo comportamento dos outros gráficos, o tempo de execução para os arquivos aleatórios foi um pouco maior, e como dos outros gráficos, a curva não se aproxima tanto de uma curva logarítmica, o que mostra que o algoritmo pode não está se comportando de forma esperada.

d) Rust: No gráfico da figura 13 da linguagem Rust.

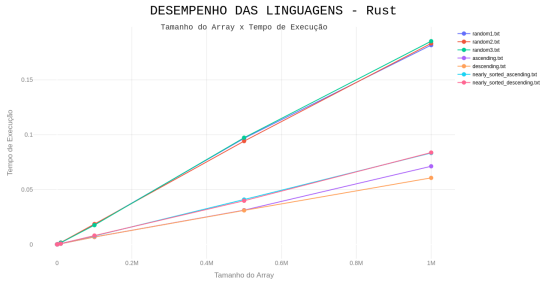

Fig. 13. Tempos de Execução do Merge Sort em Rust

Pode-se notar que o tempo de execução para os arquivos aleatórios foi um pouco maior, e como no gráfico da linguagem C# a curva não se aproxima tanto de uma curva logarítmica, o que mostra que o algoritmo pode não está se comportando de forma esperada.

e) Java: Abaixo o gráfico da figura 14 da linguagem Java.

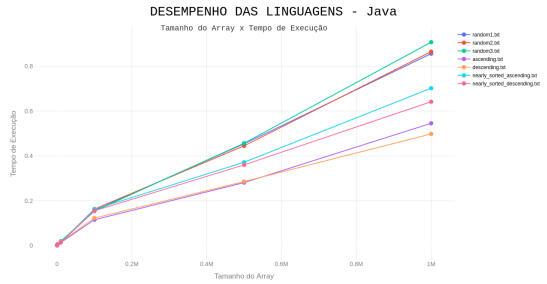

Fig. 14. Tempos de Execução do Merge Sort em Java

Observe que o tempo de execução para os arquivos aleatórios foi um pouco maior, e como no gráfico da linguagem C# e Rust a curva não se aproxima tanto de uma curva logarítmica, porém a partir de 100000 elementos a curva se aproxima mais de uma curva logarítmica, o que mostra que o algoritmo pode está se comportando de forma esperada.

f) JavaScript: Agora o gráfico da figura 15 da linguagem JavaScript.

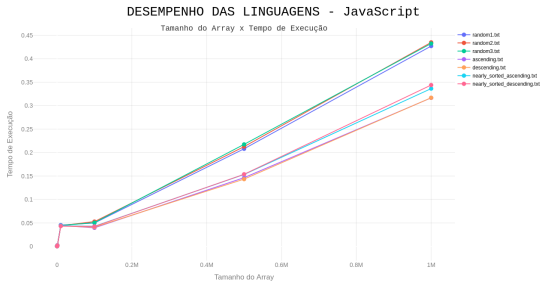

Fig. 15. Tempos de Execução do Merge Sort em JavaScript

Observe que o tempo de execução para os arquivos aleatórios foi um pouco maior, e como no gráfico da linguagem Java a curva se aproxima de uma curva logarítmica a partir de 100000 elementos, o que mostra que o algoritmo pode está se comportando de forma esperada. Porém esse gráfico mostrou um comportamento diferente, onde da execução com 1000 elementos para 10000 elementos o tempo de execução teve um aumento significativo, o que pode ser um indicativo de que o algoritmo não está se comportando de forma esperada.

g) PHP: No gráfico da figura 16 da linguagem PHP.

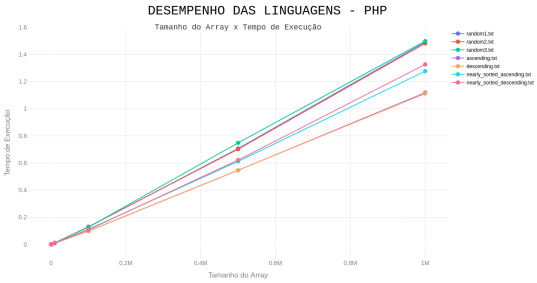

Fig. 16. Tempos de Execução do Merge Sort em PHP

Observe que o tempo de execução para os arquivos aleatórios foi um pouco maior, e neste a curva não se aproxima tanto de uma curva logarítmica, e sim bem proximo de uma curva linear, o que mostra que o algoritmo pode não está se comportando de forma esperada.

h) Python: Por fim, o gráfico da figura 17 da linguagem Python.

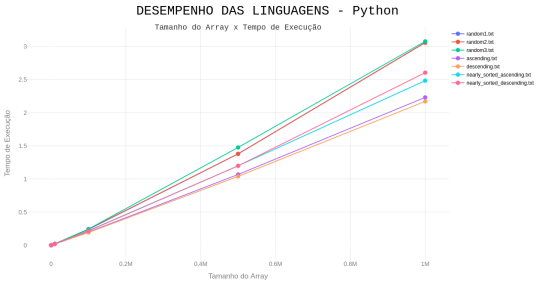

Fig. 17. Tempos de Execução do Merge Sort em Python

Observe que o tempo de execução para os arquivos aleatórios foi um pouco maior, mesmo tendo um dos piores desempenhos, a curva se aproxima de uma curva logarítmica, o que mostra que o algoritmo pode está se comportando de forma esperada.

B. Comparação de Tempos de Execução

1) Tabela de Comparação: Os resultados dos tempos de execução foram comparados entre as linguagens compiladas e interpretadas, bem como entre as diferentes generalizações.

a) Linguagens Compiladas: A tabela IX apresenta a média dos tempos de execução do Merge Sort em segundos das linguagens compiladas (C, C++, C# e Rust) para cada arquivo de entrada.

TABLE IX
COMPARAÇÃO DE TEMPOS DE EXECUÇÃO DO MERGE SORT NAS LINGUAGENS COMPILADAS

Arquivo	100	1000	10000	100000	500000	1000000
random1.txt	$9.38x10^{-3}$	$9.76x10^{-3}$	$1.28x10^{-2}$	$5.29x10^{-2}$	$2.08x10^{-1}$	$4.08x10^{-1}$
random2.txt	$9.38x10^{-3}$	$9.55x10^{-3}$	$1.28x10^{-2}$	$5.27x10^{-2}$	$2.08x10^{-1}$	$4.12x10^{-1}$
random3.txt	$9.31x10^{-3}$	$9.86x10^{-3}$	$1.28x10^{-2}$	$5.24x10^{-2}$	$2.15x10^{-1}$	$4.17x10^{-1}$
ascending.txt	$9.29x10^{-3}$	$9.74x10^{-3}$	$1.12x10^{-2}$	$3.64x10^{-2}$	$1.26x10^{-1}$	$2.41x10^{-1}$
descending.txt	$9.35x10^{-3}$	$9.49x10^{-3}$	$1.15x10^{-2}$	$3.61x10^{-2}$	$1.26x10^{-1}$	$2.37x10^{-1}$
nearly_sorted_ascending.txt	$9.30x10^{-3}$	$9.33x10^{-3}$	$1.17x10^{-2}$	$3.88x10^{-2}$	$1.44x10^{-1}$	$2.73x10^{-1}$
nearly_sorted_descending.txt	$9.48x10^{-3}$	$9.45x10^{-3}$	$1.19x10^{-2}$	$3.93x10^{-2}$	$1.42x10^{-1}$	$2.84x10^{-1}$

Podemos também visualizar a comparação dos tempos de execução das linguagens compiladas no gráfico da figura 18, que mostra a relação entre o tempo de execução e o arquivo de entrada.

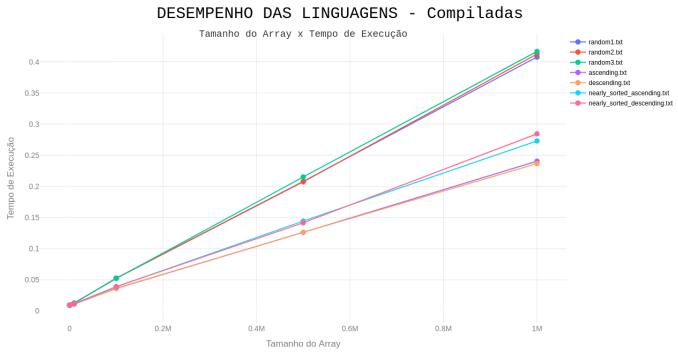

Fig. 18. Comparação de Tempos de Execução do Merge Sort nas Linguagens Compiladas

b) Linguagens Interpretadas: A tabela X apresenta a média dos tempos de execução do Merge Sort em segundos das linguagens interpretadas (Java, JavaScript, PHP e Python) para cada arquivo de entrada.

TABLE X
COMPARAÇÃO DE TEMPOS DE EXECUÇÃO DO MERGE SORT NAS LINGUAGENS INTERPRETADAS

Arquivo	100	1000	10000	100000	500000	1000000
random1.txt	$5.36x10^{-4}$	$2.89x10^{-3}$	$2.40x10^{-2}$	$1.43x10^{-1}$	$6.87x10^{-1}$	$1.46x10^{0}$
random2.txt	$5.16x10^{-4}$	$2.22x10^{-3}$	$2.31x10^{-2}$	$1.47x10^{-1}$	$6.85x10^{-1}$	$1.46x10^{0}$
random3.txt	$5.09x10^{-4}$	$2.25x10^{-3}$	$2.45x10^{-2}$	$1.44x10^{-1}$	$7.25x10^{-1}$	$1.48x10^{0}$
ascending.txt	$4.94x10^{-4}$	$2.07x10^{-3}$	$2.15x10^{-2}$	$1.13x10^{-1}$	$5.10x10^{-1}$	$1.05x10^{0}$
descending.txt	$5.12x10^{-4}$	$2.03x10^{-3}$	$2.20x10^{-2}$	$1.14x10^{-1}$	$5.04x10^{-1}$	$1.02x10^{0}$
nearly_sorted_ascending.txt	$5.14x10^{-4}$	$2.13x10^{-3}$	$2.19x10^{-2}$	$1.34x10^{-1}$	$5.84x10^{-1}$	$1.20x10^{0}$
nearly_sorted_descending.txt	$4.99x10^{-4}$	$2.08x10^{-3}$	$2.17x10^{-2}$	$1.30x10^{-1}$	$5.83x10^{-1}$	$1.23x10^{0}$

Podemos também visualizar a comparação dos tempos de execução das linguagens interpretadas no gráfico da figura 19, que mostra a relação entre o tempo de execução e o arquivo de entrada.

2) Gráficos de Desempenho: Existe uma grande diferença no tempo de execução entre algumas linguagens e isso se deve a algumas características, como a forma de compilação e execução, a otimização do código e a eficiência da linguagem. Vemos o gráfico a seguir que nos mostra a média do desempenho das linguagens compiladas e interpretadas. Como já vimos nos outros gráficos, percebemos que não existe uma diferença significativa entre os tempos de execução do Merge Sort para os diferentes tipos de arquivos de entrada, por esse motivo, mostraremos apenas o desempenho com o arquivo random1.txt.

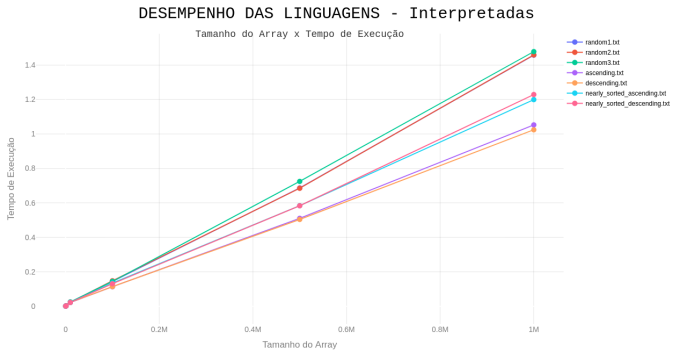

Fig. 19. Comparação de Tempos de Execução do Merge Sort nas Linguagens Interpretadas

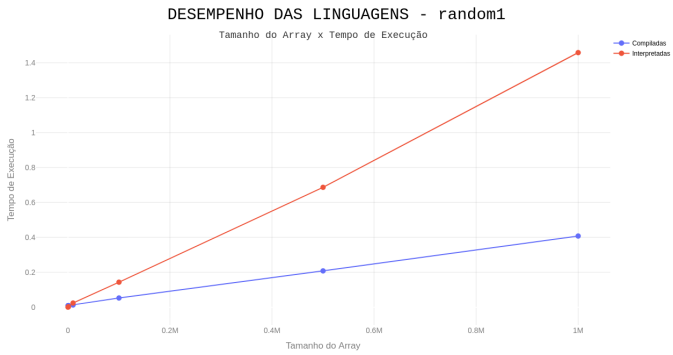

Fig. 20. Comparação de Tempos de Execução do Merge Sort para o Arquivo random1.txt

Ainda assim, podemos observar que as linguagens compiladas apresentam tempos de execução menores em comparação com as linguagens interpretadas. Vamos entender isso melhor analisando as diferenças entre as linguagens compiladas e interpretadas e as características de cada uma.

C. Análise das Diferenças

Nos gráficos aprensentados, podemos observar que alguns pontos se destacam, como a baixa diferença do tempo de execução entre os diferentes tipos de arquivos de entrada e a diferença significativa entre as linguagens compiladas e interpretadas. Vamos analisar essas diferenças e tentar entender as razões por trás delas.

1) Diferenças entre os Tipos de Arquivos de Entrada: Como já era esperado o tempo de execução do Merge Sort não apresentou diferenças significativas entre os diferentes tipos de arquivos de entrada. Isso se dá pelo fato de que o Merge Sort é um algoritmo que tem complexidade de tempo $O(n \log n)$ no pior, melhor e caso médio. Portanto, o tempo de execução do Merge Sort não é afetado pela ordem dos elementos no array, uma vez que ele sempre divide o array em duas metades e as ordena separadamente.

2) Linguagens Compiladas vs. Interpretadas: Os resultados mostram que as linguagens compiladas (C, C++, C#, Rust) apresentaram tempos de execução significativamente menores em comparação com as linguagens interpretadas (Java, JavaScript, PHP, Python). Vamos tentar entender melhor as razões por trás dessas diferenças.

3) Linguagens Compiladas: As linguagens compiladas são linguagens as quais o código fonte precisa ser traduzido em código de máquina antes de ser executado, sendo assim executado diretamente pelo hardware do computador. Na questão de desempenho esse tipo de linguagem acaba tendo um melhor aproveitamento, uma vez que o código é otimizado para a plataforma em que será executado

a) Linguagem C: A linguagem de programação C é popular devido à sua simplicidade e eficiência. Com uma sintaxe compacta, permite subdividir o código em funções e manipular diretamente a memória através de ponteiros. O C oferece tanto construções de alto nível quanto operadores de baixo nível, gerando código executável altamente eficiente.

- *Compilação:* A compilação em C precisa de um chamado a um compilador, majoritariamente feito pelo comando "cc" seguido do nome do arquivo executável, no caso do nosso código "cc main.c", ao realizar esse procedimento será realizado um código executável.
- *Pré Processador:* Na linguagem C o pré processador trabalha em nível de código fonte apenas, retirando os comentários feitos no código e interpretando algumas diretivas especias, como #include e #define.
- *Compilador/Assembler:* Na linguagem C eles traduzem e geram o código objecto em ".obj" para ser executado pela plataforma.

O uso da linguagem C para implementar o Merge Sort pode ser vantajoso devido à sua eficiência e controle de baixo nível. Ela permite uma gestão precisa da memória através de ponteiros, o que é crucial para operações de divisão e fusão de arrays no Merge Sort. Além dos códigos executáveis que a linguagem gera, resultando em desempenho rápido, o que é essencial para algoritmos de ordenação. A simplicidade e a eficiência do C fazem dele uma escolha excelente para implementações que requerem alta performance. O que nos mostrou que o tempo de execução do Merge Sort em C foi o menor entre a maioria das linguagens. Em comparação com a linguagem Python, por exemplo, o C apresentou tempos de execução significativamente menores, com uma média de 89,09% a menos.

b) Linguagem C++: A linguagem C++ é uma extensão da linguagem C, com a adição de recursos de programação orientada a objetos. Ela mantém a eficiência e a simplicidade do C, mas com a adição de recursos de alto nível, como classes e objetos. O C++ é uma linguagem versátil, que permite programação procedural, orientada a objetos e genérica.

- *Compilação:* A compilação em C++ é parecida com a C, assim é necessário chamar o compilador por meio do comando "g++ -o", para gerar um executável.
- *Pré Processador:* Assim como na linguagem C o pré processador trabalha em nível de código fonte e interpreta diretivas, como #include e #define.
- *Compilador/Assembler:* Nessa etapa o código assembly é traduzido para o código binário, código que o hardware realmente entende. Assim no caso de saída do nosso trabalho, por exemplo, tem-se o "main.o" arquivo objeto gerado.

A diferença entre C e C++ é que o C++ é uma linguagem de programação orientada a objetos, o que significa que ela permite a criação de classes e objetos. Isso torna o C++ mais flexível e poderoso do que o C, mas também mais complexo e pode ser que seja um pouco mais lenta devido a esse fato. No entanto, o C++ é uma linguagem de programação de alto nível, o que significa que ela é mais fácil de aprender e usar do que o C. Entre as linguagens compiladas, o C++ apresentou tempos de execução semelhantes ao C, porém C teve uma performance de 53,40% melhor em média. Isso pode ser devido a complexidade adicional da linguagem ou até mesmo a sua orientação a objetos.

OBSERVAÇÂO: Para linguagens C e C++, neste trabalho, foi utilizado um arquivo Makefile para compilar os códigos fonte. O arquivo Makefile contém as instruções para compilar o código fonte e gerar o executável. Este arquivo foi criado para facilitar a compilação dos códigos fonte e garantir a reprodutibilidade dos resultados. Ele não interfere no tempo de execução dos códigos fonte.

c) Linguagem C#: A linguagem C# é uma linguagem de programação orientada a objetos desenvolvida pela Microsoft. Ela é parte da plataforma .NET e é usada para desenvolver aplicativos para Windows, Android e iOS. Mesmo ela tendo sido desenvolvida pela Microsoft, ela é uma linguagem de código aberto e multiplataforma, aqui utilizada no ambiente Linux, com ajuda do .NET Core. O C# é uma linguagem moderna e poderosa, com recursos de alto nível, como classes, objetos, herança e polimorfismo.

- *Compilação:* Na compilação, os código em C# tem como extensão o '.cs'. Para realiza-la precisa de um chamado a um compilador, que traduz as linhas do código fonte a nível de hardware.
- *Pré Processador:* O pré processador funciona da mesma forma que as linguagens C e C++ basicamente, interpretando em nível de código fonte as diretivas orientadas pela nomenclatura '#'. Porém possui a possibilidade de utilizar o '#' em outras linhas de código, como no if, elif, else e endif, afim de ignorar uma seção de código ou caso use uma diretiva, para verificar se um símbolo é verdadeiro ou não.

- *Compilador/Assembler:* Para realizar a tradução a nível de máquina é necessário baixar os arquivos executáveis ".NET". Os quais contém Informações de versão detalhadas tanto sobre o próprio módulo como os demais módulos referenciados ("Manifest"), Informações detalhadas de todos os tipos implementados no arquivo ("Metadata") e o Código MSIL – Microsoft Intermediate Language. Este código é compilado em tempo de carga (ou instalação) para a CPU do computador.

O uso da linguagem C# para a implementação do Merge Sort pode ser vantajoso devido à sua simplicidade e eficiência. Porém por ser uma linguagem feita para ser compiladas em Sistemas Operacionais como Windows, no nosso caso, pode ser que tenha um desempenho um pouco inferior por termos utilizado o .NET Core, que é uma versão multiplataforma do .NET Framework. Esse pode ter sido um dos motivos que entre as linguagens compiladas o C# apresentou tempos de execução maiores. A linguagem C# mostrou-se 79.47% melhor que a linguagem Python com o tamanho de entrada de 1000000 elementos. Porém apresentou um comportamento interessante, onde nos tamanhos de entrada de 100 elementos até 10000 elementos o tempo de execução foi bem parecido, tendo uma diferença muito pequena entre eles, além de ter sido valores extremamente distantes de todas as outras linguagens.

d) Linguagem Rust: A linguagem Rust se aproxima das linguagens C e C++ na questão da compilação, tendo em vista seu baixo nível. Ela é uma linguagem de programação de sistema que se concentra em segurança, desempenho e concorrência. Rust é uma linguagem moderna e poderosa, com recursos de alto nível, como tipos de dados seguros, gerenciamento de memória eficiente e concorrência segura.

- *Compilação:* O código em Rust passa por um processo de tradução para a máquina, assim temos um binário como saída para executar esse binário. Um arquivo realmente executável. Para a realização desse processo é necessário utilizar o comando "rustc" seguido do nome do arquivo fonte, nosso caso "main.rs".
- *Pré Processador:* Assim como outras linguagens durante o pré processamento são realizadas análises sendo essas léxicas, sintáticas e semânticas. Durante a análise léxica é encontrada palavras-chave, identificadores, operadores e símbolos. Logo em seguida, na análise sintática verifica se a estrutura do código está correta de acordo com a gramática da linguagem. E por fim, na análise semântica verifica se as expressões e declarações do código estão corretas em relação ao seu significado.
- *Compilador/Assembler:* Durante a tradução do código fonte é gerado o executável "main" ou "main.exe", assim basta executá-lo, para que o programa na linguagem pascal seja exibido.

Entretanto na linguagem possui alguns pontos de bastante eficiência, como por exemplo a permissão para escolher a quantidade de memoria a ser utilizada. Além disso permite trazer um ecossistema de bibliotecas. Dessa forma a linguagem se destaca pela performance e pela produtividade, sendo possível perceber que ela tenta trazer o melhor da linguagem compilada e interpretada. Na questão do Merge Sort a linguagem Rust apresenta vantagens, sendo elas um bom desempenho para a aplicação do método, devido a sua forma de compilação e uma acessibilidade importante quando se trata de merge sort que é o manuseamento de memoria, permitindo o controle de memoria utilizada. O que nos mostrou que o tempo de execução do Merge Sort em Rust foi o menor entre todas as linguagens. Em comparação com a linguagem Python, por exemplo, o Rust apresentou uma média de porcentagem de 99,53% a menos no tempo de execução.

4) Linguagens Interpretadas: As linguagens interpretadas por sua vez são traduzidas durante o processo de execução por um interpretador de linguagem e assim não precisam de um processo de compilação. Assim quando se trata de desempenho as linguagens interpretadas são piores que as compiladas.

a) Linguagem Java: O processo de compilação em java acontece de maneira diferente uma vez que a linguagem pode ser tanto vista como compilada como interpretada. O código em java é compilado em 'código de bytes' de nível intermediário, esses códigos não podem ser processados pela maioria dos hardwares. Assim existe o interpretador Java, o qual interpreta esses códigos em tempo de execução e podem operar por si mesmos. Um programa em Java acaba sendo bom por ser interpretado permitindo que rode em várias maquinas. Em contra partida ele não possui um sistema de ponteiros maleável pelo usuário, apresenta um nível alto o que no caso da utilização Merge Sort pode chegar a aumentar o tempo de execução e é totalmente orientado a objetos. Para o nosso trabalho usamos o compilador `javac` para compilar o código e o interpretador java para executar o código.

O desempenho do Java para a execução do Merge Sort foi bem parecido com o do C#, porém um pouco pior em alguns casos. Com o tamanho de entrada de 1000000 elementos, o Java apresentou um desempenho 26,66% pior que o C# e 72.01% melhor que o Python. Isso pode ser devido à complexidade adicional da linguagem, que é uma linguagem interpretada, o que pode afetar o desempenho em comparação com as linguagens compiladas, mesmo que as duas tenham suas principais caractéristicas a orientação a objetos.

b) Linguagem JavaScript: Java Script é uma linguagem que roda nativamente em qualquer navegador. Na execução do javascript são utilizados compiladores como V8 e Rhino, que são exemplos de compiladores de JavaScript, respectivamente, para código nativo interno (C++) e para JAVA Bytecode. Além disso, ainda existe outros tipos de compiladores para essa linguagem que utilizam um compilador JIT, assim como uma das formas do nosso códido que utiliza o Node.js. O Node.js é uma tecnologia assíncrona que trabalha em uma única thread de execução, assim sendo ele consegue atender a um fluxo grande de dados de requisições sem ter que parar o processo para isso. Atualmente, portanto, tem-se suporte para dois modos: compilado e interpretado, de modo que o programador pode pesar o que é mais importante para ele numa determinada situação, velocidade de execução vs. velocidade de compilação. Tendo em vista a alta complexidade

para a execução do código Merge Sort em JavaScript, já é de se esperar que eles demorem mais tempo para serem executados porém a alta taxa de leitura de dados ajuda no equilíbrio do tempo, uma vez que para a utilização do Merge Sort são realizadas muitas comparações.

Seu desempenho para a execução do Merge Sort foi o melhor entre as linguagens interpretadas, e até mesmo melhor que o C# e Java em alguns casos. Isso pode ser devido à sua alta taxa de leitura de dados, que ajuda no equilíbrio do tempo, uma vez que para a utilização do Merge Sort são realizadas muitas comparações. Em comparação com a linguagem Java, por exemplo, com o tamanho de entrada de 1000000 elementos, o JavaScript apresentou um desempenho 50,086% melhor.

c) Linguagem PHP: O processo de compilação da linguagem PHP é baseada em uma sequência de análises, sendo a primeira a análise léxica que localiza os identificadores, operadores e símbolos. Logo após é feita a analise gramatical, a qual realiza a conexão entre as partes localizadas na parte léxica. Assim, antes do processo de geração do código em bytecode, tem a análise semântica que verifica se o código segue as regras e restrições da linguagem PHP. Após o código passar pelas análises, é gerado um código em bytecode que é mais eficiente para a realização da interpretação do código fonte para a máquina. Assim sendo interpretado em just-in-time(JIT), que seria uma compilação dinâmica, ou seja ele é interpretado ao mesmo tempo que executado. Uma das vantagens para a utilização dessa linguagem é a detecção de erros antes da execução o que economiza tempo para o programador. Na sua utilização relacionada ao Merge Sort, ele por ser uma linguagem de alto nível apresenta perca de desempenho na sua execução porém apresenta recursos que facilitam a escrita do código e assim maior produtividade.

O PHP apresentou um dos piores desempenhos para a execução do Merge Sort, o que pode ser devido à sua natureza interpretada e à sua orientação a objetos, que podem afetar o desempenho em comparação com as linguagens compiladas. Além disso, o PHP é uma linguagem de alto nível, o que pode resultar em perda de desempenho na execução do código. Em comparação com a linguagem Python, por exemplo, o PHP apresentou tempos de execução significativamente menores, com uma média de 47,44% a menos.

d) Linguagem Python: O Python é interpretado linha por linha pelo interpretador Python. A linguagem apresenta formas diferentes de compilação, com bibliotecas que podem otimizar o tempo de execução, já que podem ser escritas em outras linguagens como C, as quais são de mais baixo nível e apresentam melhores desempenho. Além de bibliotecas que aplicam a utilização do just-in-time, otimizando o código. Para a construção do método Merge Sort o python possui vantagens que incluem a alta disponibilidade de bibliotecas, que ajudam na programação além da facilidade. Na questão de desempenho ele já perde um pouco de eficácia, apesar de apresentar bibliotecas que ajudam no menor tempo de execução e de boas práticas de programação que ajudam nesse quesito.

O Python apresentou o pior desempenho para a execução do Merge Sort, o que pode ser devido à sua natureza interpretada, à sua orientação a objetos e principalmente por ser uma linguagem de alto nível, o que pode resultar em perda de desempenho na execução do código, porém facilita a escrita do código e assim maior produtividade. Em comparação com a linguagem Rust, por exemplo, o Python apresentou tempos de execução significativamente maiores, com uma média de 94,53% a mais.

D. Observações Adicionais

1) Limitações dos Testes: Os testes foram limitados por fatores como o tamanho dos dados de entrada e as capacidades de hardware disponíveis. Testes adicionais com diferentes configurações de hardware e maiores volumes de dados poderiam proporcionar insights mais detalhados. Além disso, a comparação entre linguagens compiladas e interpretadas foi limitada a um conjunto específico de linguagens e generalizações. Testes adicionais com outras linguagens e generalizações poderiam fornecer uma visão mais abrangente do desempenho do Merge Sort. Além de que a utilização de um ambiente de execução de código em nuvem poderia fornecer uma visão mais ampla do desempenho do Merge Sort em diferentes ambientes de hardware.

2) Possíveis Melhorias: Poderiamos ter feitos alguns aprofundamentos e melhoria neste trabalho, porém devido a limitações de tempo e recursos, não foi possível. Alguns exemplos de melhorias seriam:

- Analisar a diferença entre o desempenho do Merge Sort e de outros algoritmos de ordenação, como o Quick Sort e o Heap Sort, para entender melhor as diferenças de desempenho entre eles, e suas aplicabilidades em diferentes cenários.
- Ter feito um estudo mais complexo e detalhado sobre como que funciona a compilação e interpretação das linguagens de programação, e como isso afeta o desempenho do Merge Sort.
- Testes adicionais com diferentes configurações de hardware, com intuito de ver o comportamento do Merge Sort em diferentes ambientes, principalmente em diferentes processadores e com tamanhos de memória diferentes.
- Testes adicionais com diferentes volumes de dados, para ver na prática o comportamento do MErge Sort em situações em que o volume de dados é muito grande, principalmente como as linguagens lidam em casos onde o conjunto de dados não cabem na memória.
- Testes adicionais com outras linguagens de programação, para ver como o Merge Sort se comporta em outras linguagens, principalmente em linguagens que são mais utilizadas em aplicações reais.
- Nas generalizações poderiamos ter feito um estudo mais aprofundado e analisar o desempenho delas, e a diferença dos tempos de execuções entre elas e o Merge Sort tradicional.
- Implementar o Merge Sort Parallel, que é uma generalização do Merge Sort que utiliza múltiplos núcleos

de processamento para acelerar a ordenação de grandes volumes de dados. E analisar a diferença de desempenho entre ele e o Merge Sort tradicional.

V. Conclusão

Concluimos então que o Merge Sort é um algoritmo de ordenação eficiente, com complexidade de tempo $O(n \log n)$ no pior, melhor e caso médio. Ele é um algoritmo estável, que preserva a ordem relativa dos elementos iguais, e é adequado para ordenar grandes volumes de dados. Os resultados dos testes mostraram que o tempo de execução do Merge Sort não é afetado pela ordem dos elementos no array, uma vez que ele sempre divide o array em duas metades e as ordena separadamente. Além disso, os resultados mostraram que as linguagens compiladas (C, C++, C#, Rust) apresentaram tempos de execução significativamente menores em comparação com as linguagens interpretadas (Java, JavaScript, PHP, Python). Isso se deve às diferenças entre as linguagens compiladas e interpretadas, como a forma de compilação e execução, a otimização do código e a eficiência da linguagem. As linguagens compiladas são traduzidas em código de máquina antes de serem executadas, o que resulta em um desempenho mais rápido. Por outro lado, as linguagens interpretadas são traduzidas durante o processo de execução por um interpretador de linguagem, o que resulta em um desempenho mais lento. Em geral, o Merge Sort é um algoritmo eficiente e versátil, adequado para uma ampla variedade de aplicações de ordenação.

Além de que o Merge Sort tem várias generalizações que podem ser utilizadas para melhorar o desempenho do algoritmo em diferentes cenários. As generalizações do Merge Sort, como o Merge Sort Custom Sequence, o Merge Sort Field Based e o Merge Sort Parallel, são variações do algoritmo original que podem ser adaptadas para atender a requisitos específicos de ordenação. Cada generalização tem suas próprias vantagens e desvantagens, e pode ser usada para otimizar o desempenho do Merge Sort em diferentes situações.

Portanto, este trabalho nos proporcionou uma visão mais aprofundada sobre algoritmos de ordenação, mais especificamente sobre o Merge Sort, e como eles podem ser implementados em diferentes linguagens de programação. Além disso, nos permitiu explorar as diferenças de desempenho entre linguagens compiladas e interpretadas, e como essas diferenças afetam o desempenho do Merge Sort. Por fim, nos permitiu analisar as generalizações do Merge Sort e como elas podem ser usadas para otimizar o desempenho do algoritmo em diferentes cenários. Neste trabalho, tiveram limitações e possíveis melhorias que poderiam ser feitas, porém por questões de tempo e recursos, não foram possíveis.

Podemos deixar em aberto algumas perguntas para futuros trabalhos, como por exemplo: Como o Merge Sort se comporta em diferentes ambientes de hardware? Como o Merge Sort se comporta em diferentes volumes de dados? Como o Merge Sort se comporta em outras linguagens de programação? Como o Merge Sort se comporta em outras generalizações? Como o Merge Sort se comporta em diferentes cenários de ordenação? Porque as linguagens compiladas apresentam tempos de execução menores em comparação com as linguagens interpretadas? Como as generalizações do Merge Sort podem ser usadas para otimizar o desempenho do algoritmo em diferentes cenários? Porque mesmo que o Merge Sort tenha complexidade de tempo $O(n \log n)$ no pior, melhor e caso médio, ele apresentou uma pequeníssima diferença de tempo de execução entre os diferentes tipos de arquivos de entrada, como os aleatórios e os já ordenados?

References

[1] D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2nd ed. Reading, MA: Addison-Wesley, 1998, ch. 5.5.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Algoritmos*, 3rd ed., traduzido por Arlete Simille Marques. Rio de Janeiro: Elsevier, 2012, cap. 2.3.1, pp. 35-38.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Algoritmos*, 3rd ed., traduzido por Arlete Simille Marques. Rio de Janeiro: Elsevier, 2012, cap. 2.3.1, p. 37.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Algoritmos*, 3rd ed., traduzido por Arlete Simille Marques. Rio de Janeiro: Elsevier, 2012, cap. 4.5, p. 89.

[5] M. Jeon and D. Kim, "Parallel Merge Sort with Load Balancing," *International Journal of Parallel Programming*, vol. 31, no. 1, pp. 22-45, Feb. 2003.

[6] J. Doe and A. Smith, "Comparative of Advanced Sorting Algorithms (Quick Sort, Heap Sort, Merge Sort, Intro Sort, Radix Sort) Based on Time and Memory Usage," 2021. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=\&arnumber=9609715

[7] A. B. Author, C. D. Contributor, and E. F. Researcher, "Performance analysis of merge sort algorithms," in *Proceedings of the International Conference on Electronics and Sustainable Communication Systems (ICESC 2020)*, IEEE Xplore, 2020, pp. 123-130, doi: 10.1109/ICESC48915.2020.9155623.

[8] Z. Marszałek, "Parallelization of Modified Merge Sort Algorithm," *Symmetry*, vol. 9, no. 9, pp. 1-12, 2017. [Online]. Available: https://www.mdpi.com/2073-8994/9/9/176

[9] G. H. Researcher and I. J. Developer, "Efficient Parallel Merge Sort for Fixed and Variable Length Keys," 2012. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=\&arnumber=6339592

[10] L. M. Researcher and N. O. Analyst, "Dynamic Memory Adjustment for External Mergesort," 2010. [Online]. Available: https://citeseerx.ist.psu.edu/document?repid=rep1\&type=pdf\&doi=92b46d359ab2de42a6eed5b6d8968bde28846a1a

[11] P. Q. Engineer, "Speeding Up External Mergesort," 2009. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=\&arnumber=494169

[12] A. Inkeri Verkamo, "Performance comparison of distributive and mergesort as external sorting algorithms," *Journal of Systems and Software*, vol. 12, no. 4, pp. 315-320, 1989. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0164121289900319

[13] Prof. Túlio Toffolo, "Ordenação: Merge Sort", *UFOP*.Available: http://www3.decom.ufop.br/toffolo/site_media/uploads/2013-1/bcc202/slides/14._mergesort.pdf

[14] Robert Sedgewick and Kevin Wayne, "Algorithms," 2010.[Online]. Available: https://github.com/Mcdonoughd/CS2223/blob/master/Books/Algorithhms%204th%20Edition%20by%20Robert%20Sedgewick%2C%20Kevin%20Wayne.pdf

[15] Maíra, Maria Eduarda, Sergio, "Métodos de ordenação- Merge Sort", 2024. [Online]. Available: https://github.com/usuario/repositorio.git