

Merge Sort

Alícia Lopes – GU3026558

Gabriel Araújo – GU3027261

Guilherme Correa – GU3026647

Leonardo dos Reis – GU3027287

Tópicos

01

Visão Geral

Criação, algoritmo, divisão e conquista.

02

Merge e Merge Sort

Análise e algoritmo.

03

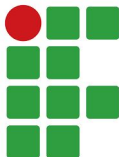
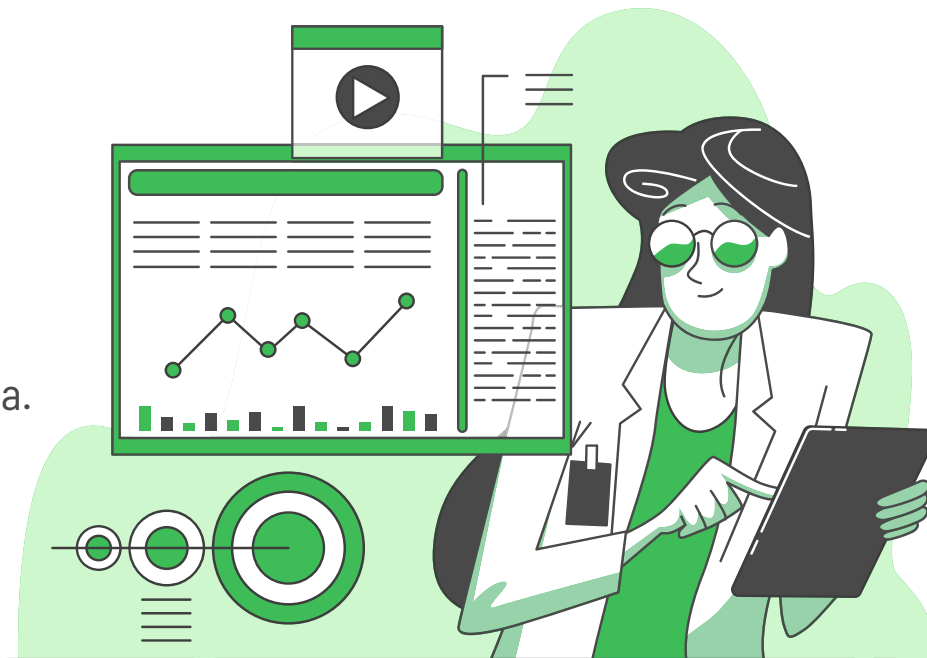
Análises

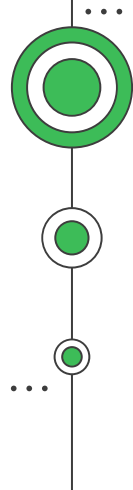
Complexidade, análise de execução e uso de memória.

04

Aplicação

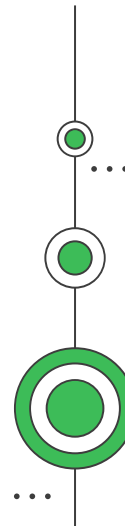
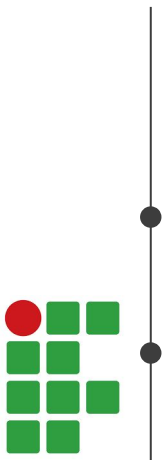
Código completo.





01

Visão Geral





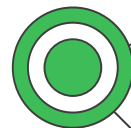
Criação

O Merge Sort é um algoritmo de ordenação recursivo (intercalação/fusão) cujo mérito pela criação é atribuído ao físico e matemático **John von Neumann**.

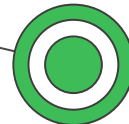
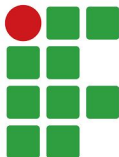
...

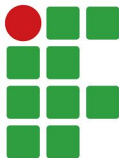


Ordenação por comparação: Merge Sort



- O algoritmo Mergesort usa a estratégia da **divisão e conquista**.
- A ideia é **dividir** o vetor em **dois subvetores**, cada um com metade dos elementos do vetor original.
- Quando os subvetores têm apenas um elemento (caso base), a recursão para. Então, os subvetores ordenados são **fundidos** (ou intercalados) num único vetor ordenado.





Exemplo

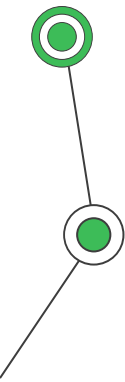
Como exemplo, ordenaremos o vetor $[5, 2, 7, 6, 2, 1, 0, 3, 9, 4]$. Inicialmente, dividimos o vetor em dois subvetores, cada um com metade dos elementos do vetor original.

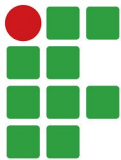
5	2	7	6	2	1	0	3	9	4
---	---	---	---	---	---	---	---	---	---

5	2	7	6	2	1	0	3	9	4
---	---	---	---	---	---	---	---	---	---

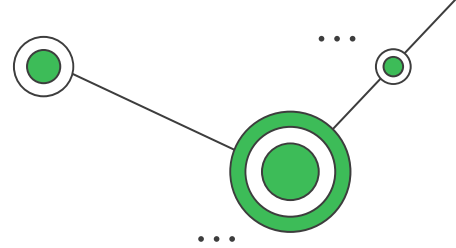
Reaplicamos o método aos dois subvetores

5	2	7	6	2	1	0	3	9	4
---	---	---	---	---	---	---	---	---	---





Exemplo

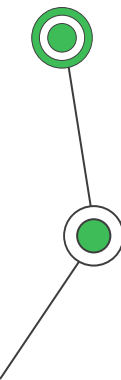


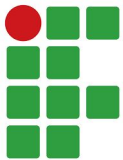
De novo,

5	2	7	6	2	1	0	3	9	4
---	---	---	---	---	---	---	---	---	---

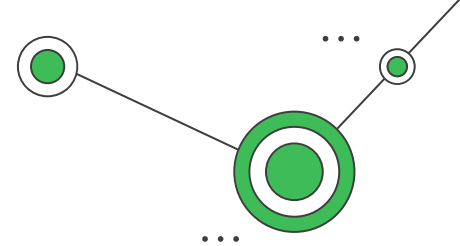
Mais uma vez, pois ainda não alcançamos o caso base em alguns subvetores

5	2	7	6	2	1	0	3	9	4
---	---	---	---	---	---	---	---	---	---





Exemplo

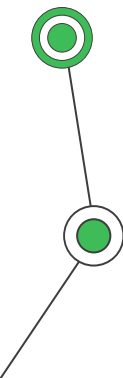


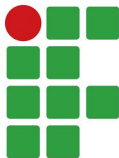
Finalmente, fazemos a fusão dos subvetores

2	5	7	2	6	0	1	3	4	9
---	---	---	---	---	---	---	---	---	---

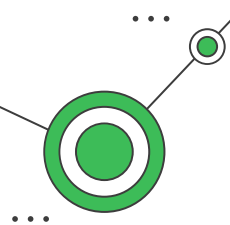
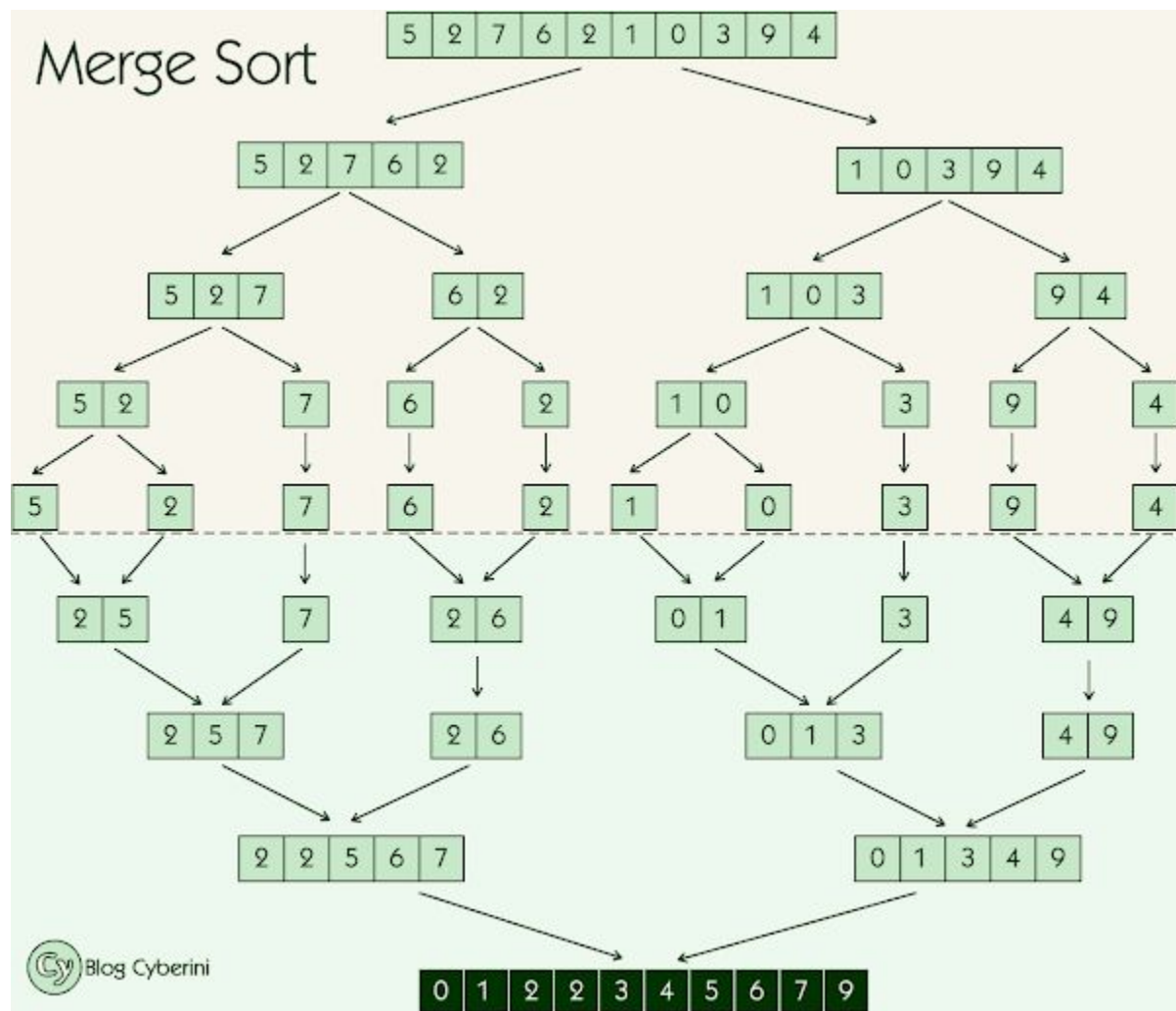
2	2	5	6	7	0	1	3	4	9
---	---	---	---	---	---	---	---	---	---

0	1	2	2	3	4	5	6	7	9
---	---	---	---	---	---	---	---	---	---





Merge Sort



Merge Sort: Ordenação por Intercalação

01

Dividir

Dividimos a lista de tamanho n em duas sub-listas de tamanho aproximadamente iguais (de tamanho $n/2$).

02

Conquistar

Com as duas sub-listas ordenadas, construímos uma lista ordenada de tamanho n ordenado.

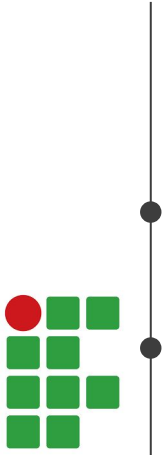
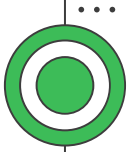
Vídeo – Dividir e Conquistar

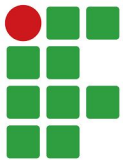
Nesse vídeo é abordado de modo mais visual a divisão e a ordenação.



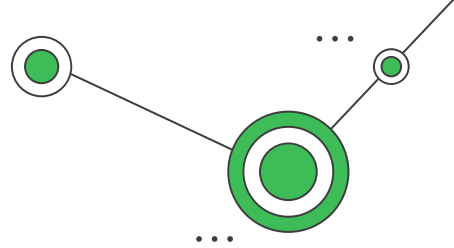
02

Merge e Merge Sort





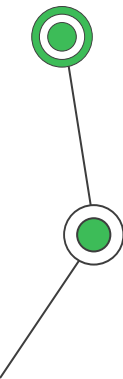
Merge Sort e Merge

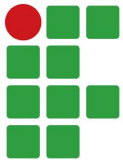


Como dito, o Merge Sort é um algoritmo de divisão-e-conquista.

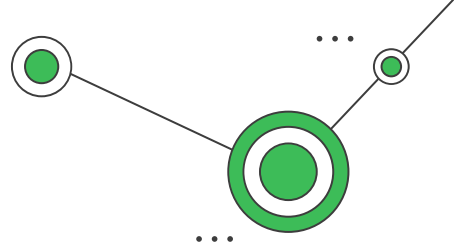
A parte da divisão, é chamada MergeSort, onde o array é “dividido” até sobrar apenas um elemento.

A parte da conquista é o Merge, isto é, a fusão de elementos divididos do array de forma ordenada.





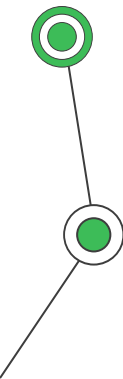
Merge Sort



Como é feita a divisão do array no MergeSort?

Basta “dividir” o array de forma recursiva na metade até que sobre apenas um elemento.

Entretanto, não dividimos de fato, não criamos dois arrays e transferimos todos os elementos, seria muito custoso. O que a gente faz é usar os índices : **início, meio e fim** para controlar as partes do array que o algoritmo deve agir.

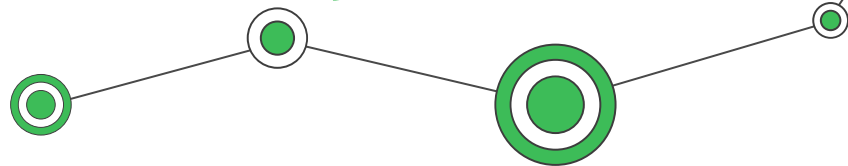


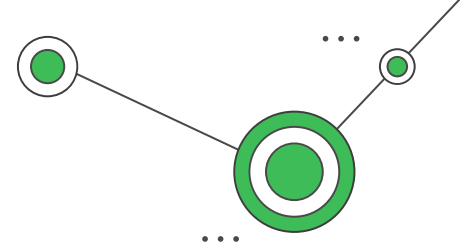
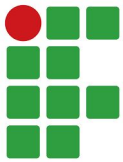
Ordenação por comparação: Merge Sort

Na prática, não queremos ficar criando arrays separados para uni-los. Isso custa **memória e processamento**, pois a cada array criado temos que transferir os elementos do array original para ele.

O que fazemos então é **organizar os dados no array** a ser ordenado de forma que uma parte dele esteja ordenada e outra também.

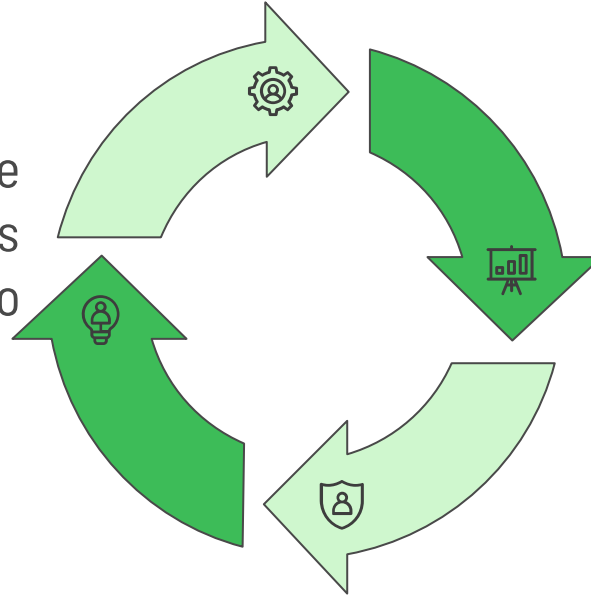
Assim, no Merge Sort não fazemos o merge de dois arrays, mas fazemos o **merge de duas partes ordenadas de um mesmo array**.



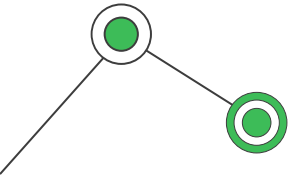


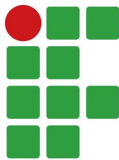
Merge

Merge é uma rotina que combina dois arrays ordenados em um outro também ordenado.

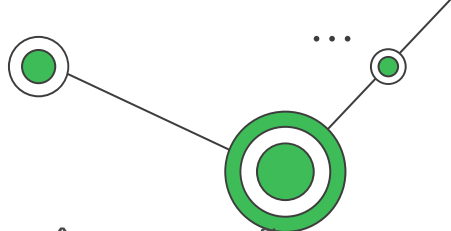


Ou seja, o Merge Sort aplica o Merge várias vezes para ordenar um array.





Implementação do Merge Sort



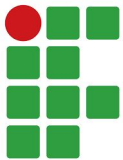
Em primeiro lugar, vamos analisar a assinatura do método. Os parâmetros são o próprio array a ser ordenado, um índice **inicio** e um índice **fim** que delimita a porção do array que o algoritmo deve analisar. Na primeira chamada, temos que **inicio = 0** e **fim = (sizeof(vetor) / sizeof(vetor[0])) - 1**.

```
void mergeSort(int v[], int inicio, int fim) {
```

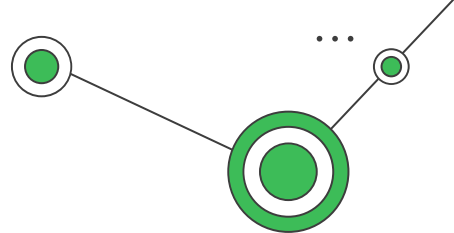


Seguindo. Na primeira linha do método, temos a condição de parada do algoritmo (**inicio >= fim**). Isto é, quando a porção do algoritmo a ser analisada possui apenas um elemento, não há mais a necessidade de “quebrá-lo”.

```
if (inicio >= fim) return;
```



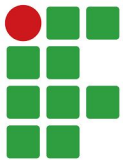
Implementação do Merge Sort



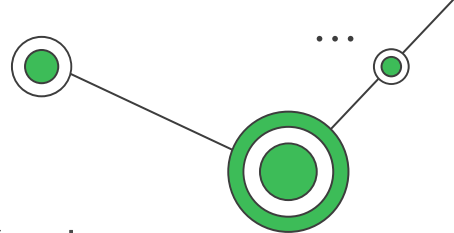
Caso ainda seja necessário “quebrar” o array (if left < right), a primeira linha define **meio** como sendo o valor central entre **início** e **fim**. A segunda e a terceira são chamadas recursivas para a metade da esquerda (**de início até meio**) e para a metade da direita (**de meio + 1 até fim**). Por fim, após cada quebra há uma chamada ao método merge (para a ordenação), passando os limites a serem considerados (**início, meio e fim**).



```
int meio = (inicio + fim) / 2;
    mergeSort(v, inicio, meio);
    mergeSort(v, meio + 1, fim);
    merge(v, inicio, meio, fim);
```



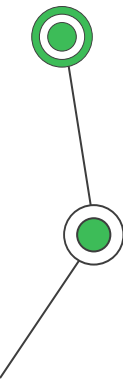
Implementação do Merge

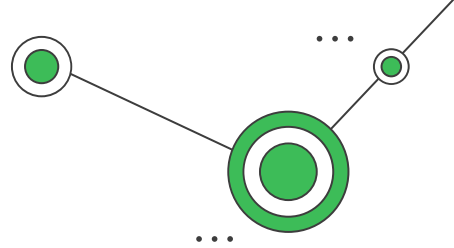
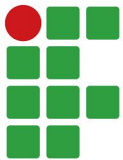


Em primeiro lugar, vamos entender a assinatura do método `merge`. Naturalmente, ele recebe como parâmetro o array a ser processado. Recebe também três índices: **início**, **meio** e **fim**, que determinam os limites em que o algoritmo deve agir.

A parte do array que é delimitada por **início** e **meio** estará ordenada e a parte do array delimitada por **meio + 1** e **fim** também estará ordenada.

início			meio				fim		
5	2	7	6	2	1	0	3	9	4



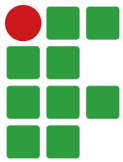


Implementação do Merge

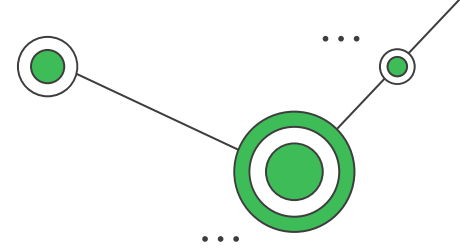
Para isso, como fazer manipulações em nosso array original, precisamos de um array auxiliar (**aux**) para guardar o estado.

```
void merge(int v[], int inicio, int meio, int fim) {  
    // transfere os elementos entre inicio e fim para um  
    array auxiliar.
```

```
    int h[fim+1];  
    for (int i = inicio; i <= fim; i++) {  
        aux[i] = v[i];  
    }
```

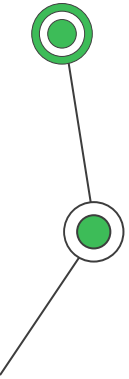


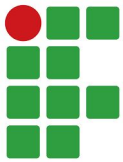
Implementação do Merge



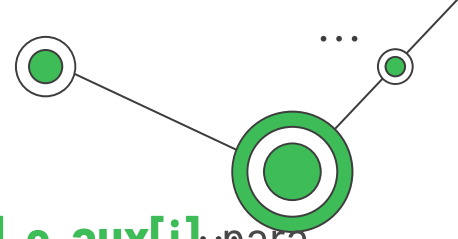
As próximas linhas definem os valores de **i**, **k** e **j** que são os índices usados para controle da execução e comparação dos elementos. **i** marca o início da primeira parte do array, **j** marca o início da segunda parte do array e **k** marca a posição em que o menor elemento entre **aux[i]** e **aux[j]** deve ser adicionado.

```
int i = fim;  
int j = meio + 1;  
int k = inicio;
```



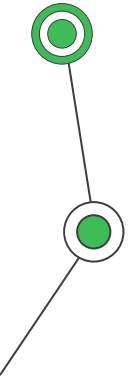


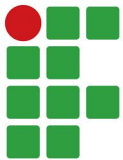
Implementação do Merge



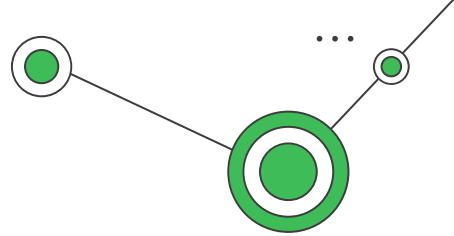
Agora, o algoritmo passa a tratar da comparação entre **aux[i]** e **aux[j]** para adicionar o menor em **vetor[k]**. Lembre-se: se **aux[i]** for menor ou igual, **vetor[k] = aux[i]** e **i** e **k** são incrementados. Caso contrário, **vetor[k] = aux[j]** e **j** e **k** são incrementados. Isso é feito até que uma das partes tenha sido completamente percorrida, isto é, se **i** atingir **meio** ou **j** atingir **fim**.

```
while (i <= meio && j <= fim) {  
    if (aux[i] <= aux[j]) {  
        v[k] = aux[i];  
        i++;  
    } else {  
        v[k] = aux[j];  
        j++;  
    }  
    k++;  
}
```





Implementação do Merge



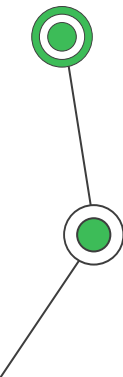
Uma das duas partes do array será consumida em sua totalidade antes da outra. Basta então, fazermos o append (adiciona um novo elemento ao final de um array) de todos os elementos da parte que não foi completamente consumida.

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

```
while (i <= meio) {  
    v[k] = aux[i];  
    i++;  
    k++;  
}
```

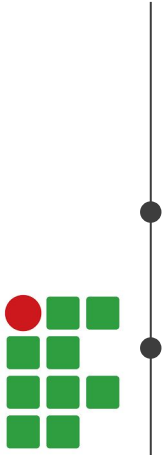
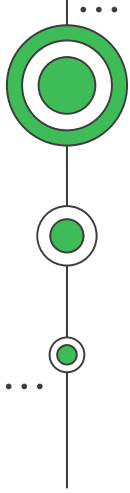
```
// se a metade final não foi toda consumida, faz o append.
```

```
while (j <= fim) {  
    v[k] = aux[j];  
    j++;  
    k++;  
}
```



03

Análise



Análise do tempo de execução

A equação de recorrência do Merge Sort é:

$$T(n) = 2T(n/2) + \Theta(n)$$

O termo **$2T(n/2)$** é o tempo para ordenar dois vetores de tamanho **$n/2$** , já o termo **$\Theta(n)$** é o tempo para fundir/intercalar esses vetores, isto é, é o tempo de método **merge**.



Explicação

O Merge Sort possui duas chamadas recursivas, cada uma reduzindo o problema (tamanho do array) na metade.

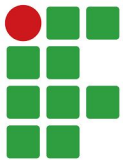
Ou seja, $2 * T(n / 2)$.

Além disso, há também uma chamada ao método Merge, que sabemos ser $\Theta(n)$.

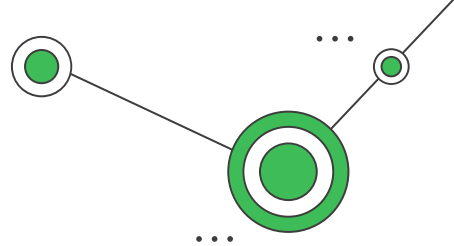


Vídeo 02

João Arthur - Prof^o UFCG

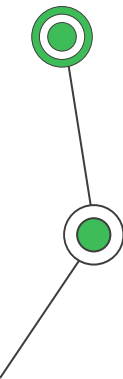


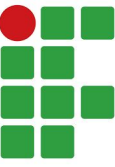
Complexidade Assintótica



Se nossa missão é ordenar um array comparando seus elementos, do ponto de vista assintótico, **$n * \log n$ é o nosso limite inferior**. Ou seja, nenhum algoritmo de ordenação por comparação é mais veloz do que $n * \log n$.
Formalmente, todos são $\Omega(n * \log n)$.

No caso do Merge Sort, uma característica importante é que sua **eficiência é $n * \log n$ para o melhor, pior e para o caso médio**. Ou seja, ele não é somente $\Omega(n * \log n)$, mas é $\Theta(n * \log n)$. Isso nos dá uma garantia de que, independente da disposição dos dados em um array, a ordenação será eficiente.





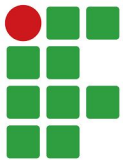
...

...

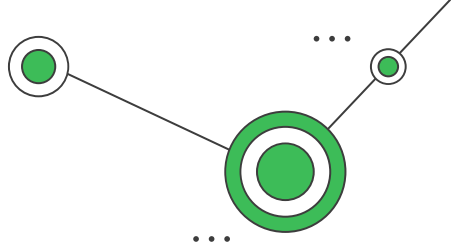
"O Merge Sort nos
garante eficiência n^*
 $\log n$ para todos os
casos"

...





Diálogo entre alunos



...

Antonio Angelo

"Porque as "quebras" do array sempre ocorrem na metade."

...

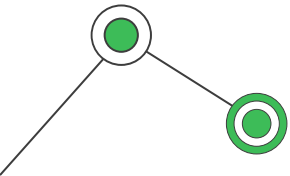
Maria Martins

"Por que o Merge Sort sempre nos garante eficiência $n * \log n$?"

...

Julio Souza

"Ou seja, independente dos dados, estamos sempre dividindo o array na metade."



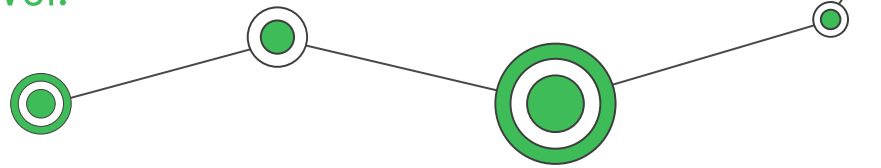
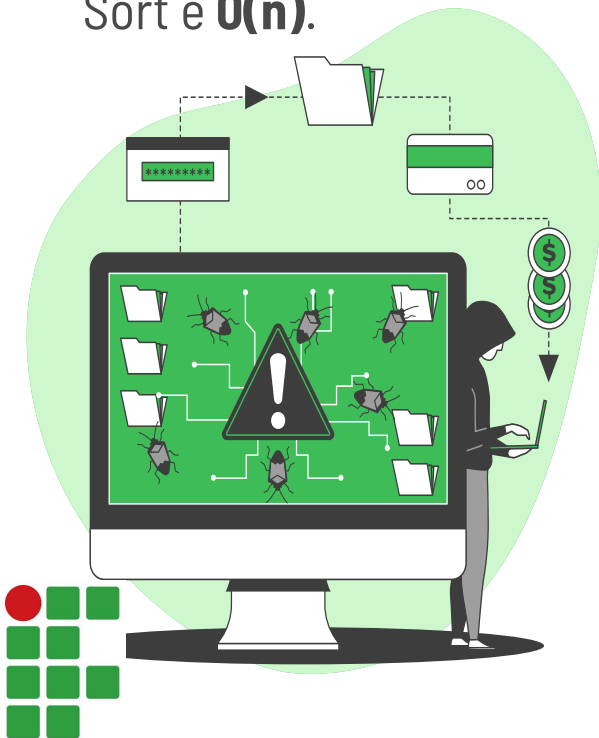
Análise do uso de memória

O Merge Sort usa um array auxiliar (**aux**) na ordenação. O tamanho de **aux** é o mesmo do array original. Ou seja, do ponto de vista de uso de memória, o Merge Sort é **$O(n)$** .

O Merge Sort não é in-place.

É importante lembrar também que a ordenação é estável, pois **mantém a ordem dos elementos iguais**. Isso porque decidimos que, se o elemento mais à esquerda for menor ou IGUAL ao mais à direita, ele deve ser colocado primeiro no array ordenado.

O Merge Sort é estável.





Vantagens do Merge Sort



01

Eficiência

Ele executa em um tempo razoável mesmo para conjuntos de dados muito grandes.

02

Flexibilidade

É facilmente adaptado para lidar com diferentes tipos de dados e diferentes formas de comparação.

03

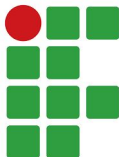
Estabilidade

Ele preserva a ordem dos elementos que têm chaves iguais.

04

Baixo consumo de memória

Não requer muita memória adicional para a sua execução.



...

Tabela comparativa do tempo de execução (em segundos) de alguns algoritmos de ordenação em C para diferentes tamanhos de array:


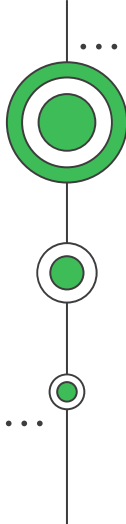
Tamanho do Array	Bubble Sort	Selection Sort	Insertion Sort	Merge Sort
10	0.000003	0.000003	0.000003	0.000002
100	0.000223	0.000058	0.000024	0.000012
1.000	0.022554	0.002172	0.000921	0.000246
10.000	2.855848	0.216389	0.096632	0.004043
100.000	N/D	N/D	N/D	0.664948

Análise da tabela

Observe que o Bubble Sort, Selection Sort e Insertion Sort têm uma complexidade assintótica de $O(n^2)$, enquanto o Merge Sort tem uma complexidade de $O(n \log n)$. Isso significa que, para tamanhos maiores de array, o **Merge Sort se torna significativamente mais rápido do que os outros algoritmos.**

Além disso, observe que para tamanhos maiores de array (100.000), o Bubble Sort, Selection Sort e Insertion Sort não puderam ser executados no meu computador devido à sua ineficiência, enquanto **o Merge Sort executou em menos de um segundo.**





04

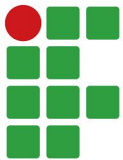
Aplicação



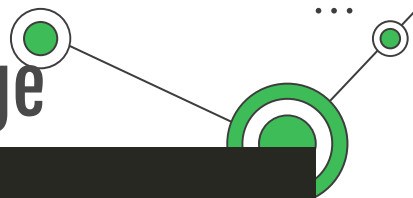
Vídeo

Antes de tudo, veremos de modo visual e sonoro a funcionalidade do Merge Sort.





Aplicação do Merge Sort – void merge



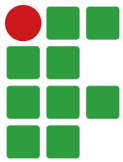
```
#include <stdio.h>

// Função de intercalação para mesclar dois subvetores em um vetor
ordenado
void merge(int arr[], int inicio, int meio, int fim) {
    int i, j, k;
    int n1 = meio - inicio + 1;
    int n2 = fim - meio;

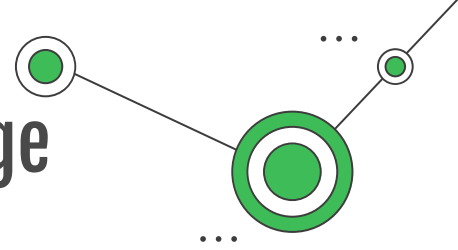
    // Cria vetores temporários para os subvetores da esquerda e
    direita
    int INICIO[n1], FIM[n2];

    // Copia os dados para os vetores temporários
    for (i = 0; i < n1; i++)
        INICIO[i] = arr[inicio + i];
    for (j = 0; j < n2; j++)
        FIM[j] = arr[meio + 1 + j];
```





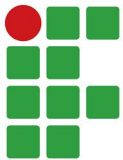
Aplicação do Merge Sort – void merge



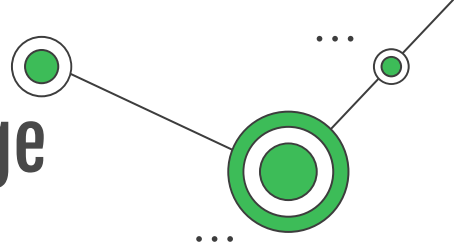
```
// Intercala os dois vetores temporários de volta ao vetor original arr[l..r]
i = 0; // Índice inicial do primeiro subvetor
j = 0; // Índice inicial do segundo subvetor
k = inicio; // Índice inicial do vetor mesclado

while (i < n1 && j < n2) {
    if (INICIO[i] <= FIM[j]) {
        arr[k] = INICIO[i]; i++;
    } else {
        arr[k] = FIM[j]; j++;
    }
    k++;
}
```





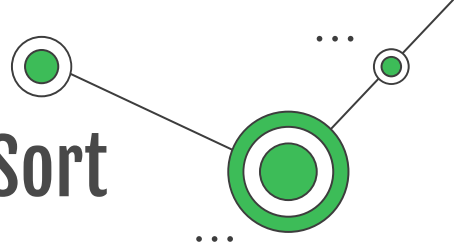
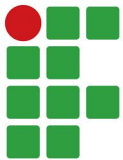
Aplicação do Merge Sort – void merge



```
// Copia os elementos restantes de L[], se houver algum
while (i < n1) {
    arr[k] = INICIO[i];
    i++;
    k++;
}

// Copia os elementos restantes de R[], se houver algum
while (j < n2) {
    arr[k] = FIM[j];
    j++;
    k++;
}
}
```



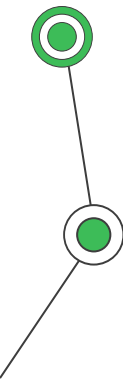


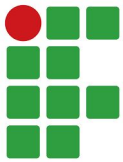
Aplicação do Merge Sort – void mergeSort

```
/Função principal que implementa o Merge Sort
void mergeSort(int arr[], int inicio, int fim) {
    if (inicio < fim) {
        // Encontra o ponto médio para dividir o array em duas
        metades
        int meio = inicio + (fim - inicio) / 2;

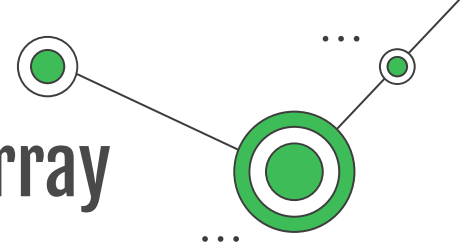
        // Ordena a primeira e a segunda metade
        mergeSort(arr, inicio, meio);
        mergeSort(arr, meio + 1, fim);

        // Mescla as duas metades ordenadas
        merge(arr, inicio, meio, fim);
    }
}
```





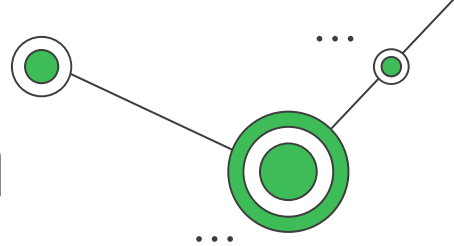
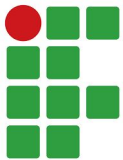
Aplicação do Merge Sort – void printArray



```
// Função auxiliar para imprimir o vetor
void printArray(int arr[], int size) {
    int i;

    for (i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
```





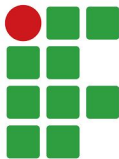
Aplicação do Merge Sort – int main

```
// Função principal
int main() {
    int arr[] = { 5, 2, 7, 6, 2, 1, 0, 3, 9, 4 };
    int arr_size = sizeof(arr) / sizeof(arr[0]);

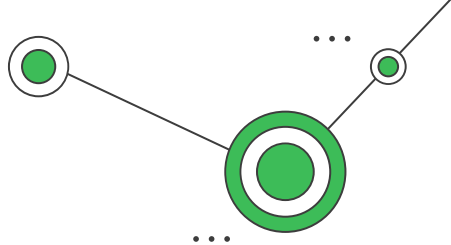
    printf("Vetor original:\n");
    printArray(arr, arr_size);
    mergeSort(arr, 0, arr_size - 1);
    printf("\nVetor ordenado:\n");
    printArray(arr, arr_size);

    return 0;
}
```



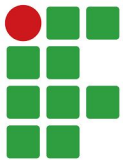


Bibliografia



- FEOFILOFF, Paulo. Análise de Algoritmos. "Ordenação MergeSort". Departamento de Ciência da Computação, Instituto de Matemática e Estatística da USP, atualizado em 16 set. 2020. Disponível em: https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/mergsrt.html. Acesso em 01 mar. 2023.
- AVILA, Sandra. Algoritmos e Programação de Computadores. "Ordenação: Merge Sort". Instituto de Computação, IC/Unicamp, MC102, publicado em jun. de 2019. Disponível em: <https://www.ic.unicamp.br/~sandra/pdf/class/2019-1/mc102/2019-06-17-MC102KLMN-Aula27.pdf>. Acesso em 01 mar. 2023.
- CORMEN, T. H. et al. Algoritmos: teoria e prática. 3 ed. Rio de Janeiro: Elsevier, 2012.
- FELIPE, Henrique. Algoritmos de Ordenação. "Merge Sort". Publicado em 01 jul. 2028. Disponível em: <https://www.blogcyberini.com/2018/07/merge-sort.html>. Acesso em 01 mar. 2023.
- AMBRÓSIO, Ana Paula; STEFANES, Marco Aurélio. Algoritmos e estruturas de dados. 1. ed. São Paulo: Novatec, 2015.





Obrigado!

Alícia Lopes
Gabriel Araújo
Guilherme Correa
Leonardo dos Reis

