



Universidade Federal do Rio Grande do Norte
Instituto Metr pole Digital

An lise de algoritmos cl ssicos de ordena  o e busca

Bianca Medeiros, Gabriel Carvalho, Marina Medeiros, Vinicius de Lima

Estrutura de Dados B sica II

Trabalho da Primeira Unidade

31 de outubro de 2024

Sumário

1	Análise Teórica	1
1.1	Bubble Sort	1
1.1.1	Bubble Sort Iterativo	2
1.1.2	Bubble Sort Recursivo	3
1.2	Merge Sort	6
1.2.1	Merge Sort Recursivo	7
1.2.2	Merge Sort Iterativo	7
1.2.3	Função auxiliar Merge	7
1.2.4	Análise de complexidade	10
1.3	Quick Sort	13
1.4	Quick Sort	13
1.4.1	Quick Sort Iterativo	13
1.4.2	Quick Sort Recursivo	13
2	Análise de Algoritmo	18
2.1	Função iterativa	18
2.2	Função recursiva	18
2.3	Implementação dos algoritmos de ordenação	18
2.3.1	Bubble Sort	18
2.3.2	Quick Sort	19
2.3.3	Merge Sort	20
2.3.4	Organização do projeto e corretude do algoritmo	22
2.4	Análise de performance dos algoritmos de ordenação	24

Capítulo 1

Análise Teórica

1.1 Bubble Sort

O Bubble Sort, também conhecido como "Ordenação por bolha" ou "Ordenação por flutuação", é um dos algoritmos de ordenação mais simples.

Neste algoritmo, são realizadas comparações entre os dados armazenados em um vetor de tamanho n . Cada elemento na posição i é comparado com o elemento na posição $i + 1$. Quando a ordenação procurada — seja ela crescente ou decrescente — é encontrada, ocorre uma troca de posições entre os elementos.

O algoritmo executa dois laços principais:

1. O primeiro laço percorre a quantidade de elementos do vetor:

```
for (j = 1; j <= n; j++)
```

2. O segundo laço, que está dentro do primeiro, percorre da primeira à penúltima posição do vetor:

```
for (i = 0; i < n - 1; i++)
```

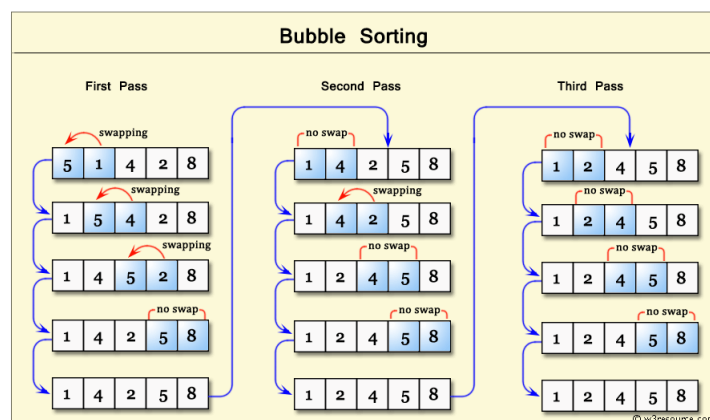


Figure 1.1: Diagrama exemplo de um Bubble Sort

1.1.1 Bubble Sort Iterativo

Algorithm 1 Bubble Sort

Input: Lista $A = A_1, A_2, \dots, A_n$
Output: Lista A ordenada

```

1: function BubbleSort( $A$ )
2:   for  $j \leftarrow 1$  to  $n - 1$  do
3:     for  $i \leftarrow 0$  to  $n - 2$  do
4:       if  $A[i] > A[i + 1]$  then
5:          $aux \leftarrow A[i]$ 
6:          $A[i] \leftarrow A[i + 1]$ 
7:          $A[i + 1] \leftarrow aux$ 
8:       end if
9:     end for
10:  end for
11: end function

```

Análise

Nesse algoritmo, o fator relevante que determina o seu tempo de execução é o número de comparações realizadas. Considerando que o algoritmo foi implementado para um vetor com n posições, o número de iterações do primeiro laço é n .

O segundo laço possui $n - 1$ iterações, mas como ele está interno ao primeiro, ele será executado $n(n - 1) = n^2 - n$ vezes. Portanto, podemos dizer que o tempo de execução do algoritmo Bubble Sort, em sua forma iterativa, será $O(n^2)$, pois:

$$\lim_{n \rightarrow \infty} \frac{n^2 - n}{n^2} = \lim_{n \rightarrow \infty} \frac{n - 1}{n} = \lim_{n \rightarrow \infty} 1 - \frac{1}{n} = 1 \in \mathbb{R}_+^*.$$

De forma análoga, podemos dizer que o Bubble Sort será $\Omega(n^2)$ e $\Theta(n^2)$.

Nesse algoritmo, não há situações melhores ou piores. O comportamento do algoritmo não mudará, independentemente do valor de entrada. Ele realizará todas as comparações, mesmo que desnecessárias.

1.1.2 Bubble Sort Recursivo

Algorithm 2 Bubble Sort Recursivo

Input: Lista $A = A_1, A_2, \dots, A_n$
Output: Lista A ordenada

```

1: function BubbleSortRecursivo( $A, n$ )
2:   if  $n == 1$  then return
3:   end if
4:   for  $j \leftarrow 0$  to  $n - 1$  do
5:     if  $A[j] > A[j + 1]$  then
6:        $aux \leftarrow A[j]$ 
7:        $A[j] \leftarrow A[j + 1]$ 
8:        $A[j + 1] \leftarrow aux$ 
9:     end if
10:  end for
11:  BubbleSortRecursivo( $A, n - 1$ )
12: end function

```

Análise

Agora, faremos a análise da complexidade da versão recursiva do algoritmo Bubble Sort, utilizando os quatro métodos estudados na disciplina:

1. Substituição:

Com esse método, mostraremos, por indução, que $T(n) = T(n - 1) + n$ é limitada por $f(n) = n^2$.

- **Caso base:** $n = 1$.
Sabemos que $T(1) = 1$ e $f(1) = 1$.
Logo, $T(1) \leq f(1)$.
Portanto, a base da indução é válida.
- **Hipótese de Indução:** Suponha que $T(k) \leq k^2$ para todo $1 < k \leq n$.
- **Passo indutivo:** Queremos provar que $T(n + 1) \leq (n + 1)^2$.
Observe que:

$$T(n) \leq n^2 \quad (\text{pela hipótese de indução})$$

$$T(n) + n \leq n^2 + 2n \quad (\text{pois } n \leq 2n)$$

$$T(n) + n + 1 \leq n^2 + 2n + 1$$

$$T(n + 1) \leq (n + 1)^2$$

Portanto, para todo k tal que $1 < k \leq n$, $T(n) \leq f(n) = n^2$.

Assim, $T(n)$ é $O(n^2)$.

2. Método Mestre:

Esse método permite resolver recorrências da forma $T(n) = a T\left(\frac{n}{b}\right) + \Theta(n^k)$, com $a \geq 1$, $b > 1$, e $k \geq 0$, onde a , b , e k são constantes.

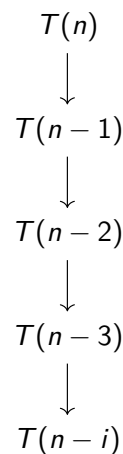
Como não é possível escrever $T(n) = T(n-1) + n$ nesse formato, sem que b seja uma função de n , não podemos aplicar o Método Mestre a essa recorrência.

3. Árvore Recursiva:

Esse método consiste em desenhar uma árvore cujos nós representam os tamanhos dos problemas correspondentes. Cada nível i contém todos os subproblemas de profundidade i . Dois aspectos importantes devem ser considerados: a altura da árvore e o número de passos executados em cada nível. A solução da recorrência, que é o tempo de execução do algoritmo, é a soma de todos os passos de todos os níveis:

$$\sum_{i=0}^{\text{níveis}} (\text{tempo por nó}) \cdot (\text{quantidade de nós})$$

Precisamos saber a quantidade de níveis, ou seja, o valor que corresponde à altura da árvore recursiva. Nesse caso, temos a seguinte árvore recursiva:



A partir da árvore de recorrência, podemos formar a seguinte tabela:

Nível da Árvore	Tamanho da Entrada	Custo por Nó	Quantidade de Nós
0	n	n	1
1	$n-1$	$n-1$	1
2	$n-2$	$n-2$	1
3	$n-3$	$n-3$	1
\vdots	\vdots	\vdots	\vdots
i	$n-i$	$n-i$	1

Com isso, podemos concluir que a soma de todos os passos executados por todos os níveis é dada por:

$$T(n) = 1 + 2 + 3 + \dots + (n-2) + (n-1) + n = \frac{(n+1)n}{2} = \frac{n^2 + n}{2}$$

Portanto, temos:

$$T(n) = O(n^2)$$

4. Iteração:

Aqui, expandiremos a recorrência até o caso base.

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= T(n-2) + (n-1) + n \\ &= T(n-3) + (n-2) + (n-1) + n \\ &\quad \vdots \\ &= T(n-i) + (n-i+1) + (n-i+2) + \cdots + (n-1) + n \end{aligned}$$

O caso base ocorrerá quando $i = n - 1$. Nesse momento, teremos:

$$T(n) = T(1) + 2 + 3 + \cdots + (n-1) + n$$

A soma $2 + 3 + \cdots + n$ é uma progressão aritmética, e pode ser calculada por:

$$T(n) = T(1) + \frac{(2+n)(n-1)}{2}$$

Assim, substituindo $T(1)$ por uma constante c , obtemos:

$$T(n) = c + \frac{n^2 + n - 2}{2}$$

Como c é uma constante, podemos ignorá-la na análise assintótica e dizer que:

$$T(n) = O(n^2)$$

1.2 Merge Sort

Desenvolvido por [Jon Von Neumann](#) em 1945, o *Merge Sort* é um algoritmo de [dividir para conquistar](#), que subdivide uma lista em singletons e os mescla em sublistas ordenadas até que exista apenas uma sublista, esta sublista é a lista original ordenada. Imagine o seguinte caso:

Você tem um baralho de cartas e gostaria de organizá-lo, seguindo o conceito do *Merge Sort* você trabalha da seguinte forma:

- **Divisão:** Primeiro você divide o baralho em dois baralhos menores. Cada um desses baralhos é dividido novamente até que cada sub-baralho tenha uma carta.
- **"Merge"(Mescla):** Nesse momento, com cada sub-baralho com uma carta, todos estão ordenados, então você começa a juntar os sub-baralhos comparando duas cartas de cada baralho e colocando-as em ordem crescente. Assim dois grupos de uma carta se tornam um baralho de duas cartas ordenadas. Em seguida, dois baralhos de duas cartas são mesclados para formar um baralho de quatro cartas, e assim por diante, até que todas as cartas estejam combinadas novamente, mas agora ordenadas.

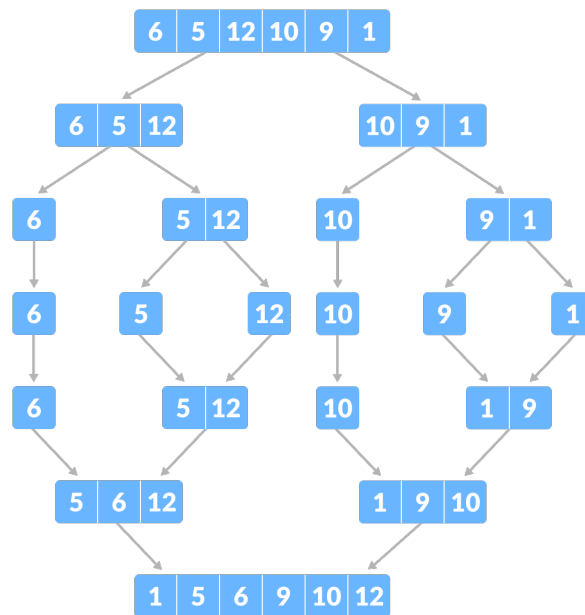


Figure 1.2: Diagrama exemplo de um Merge Sort

1.2.1 Merge Sort Recursivo

Dito isso, agora fica mais fácil estabelecer o pseudocódigo na forma recursiva. O caso base será se a lista tem no máximo elemento, pois já está ordenada. No caso recursivo, pelo teorema da recursão temos acesso ao "caso anterior", ou seja a primeira metade e segunda metade da lista original ordenadas, portanto, podemos mesclá-las.

Input: $\text{lista} = x_0, x_1, \dots, x_{N-1}$

```
1: function MergeSort(lista)  
2:   if tamanho da lista  $\leq 1$  then return  
3:   end if  
4:   return Merge(MergeSort(1ª metade da lista), MergeSort(2ª metade da lista)  
5: end function
```

1.2.2 Merge Sort Iterativo

TODO

1.2.3 Função auxiliar Merge

Nos resta agora apenas definir a função **Merge**, que vai ser responsável por mesclar duas listas que estão ordenadas. Para tal, a função deve comparar um elemento da primeira lista com um da segunda e anexar o menor elemento entre os dois no final da lista resultado. Por exemplo:



10	50	75	80
----	----	----	----

55	67	79	90
----	----	----	----

Figure 1.3: Comparando o primeiro elemento da primeira lista com o primeiro da segunda

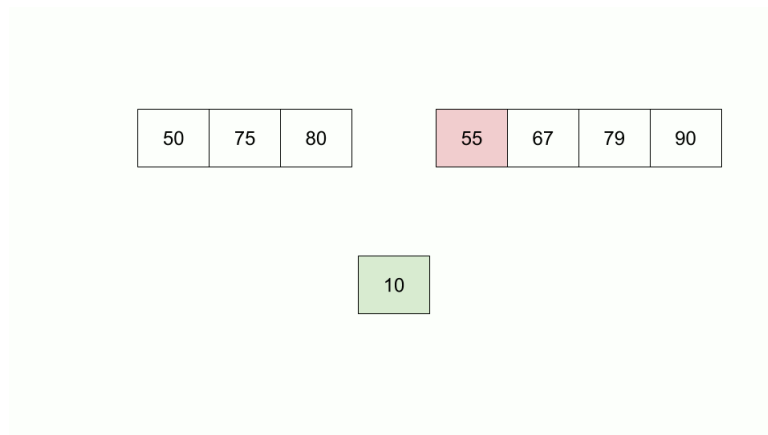


Figure 1.4: 10 é menor que 55, então é anexado no final da lista resultado

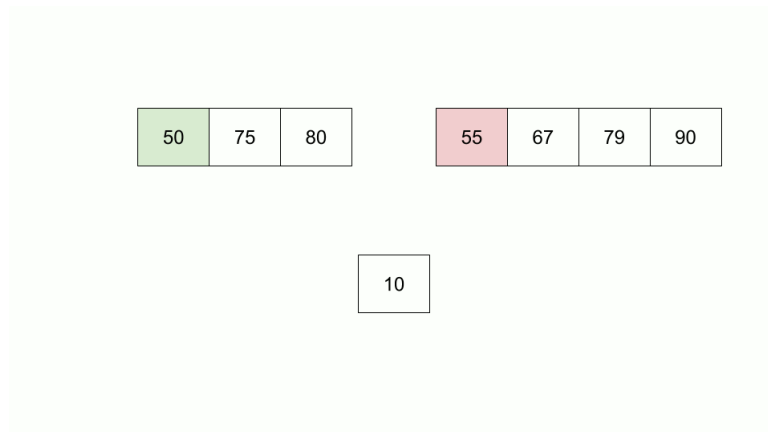


Figure 1.5: Comparando o próximo elemento da primeira lista

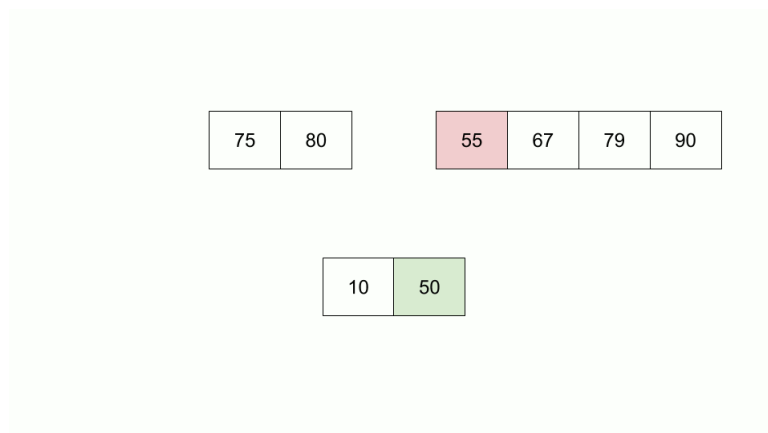


Figure 1.6: 50 é menor que 55, então é anexado no final da lista resultado

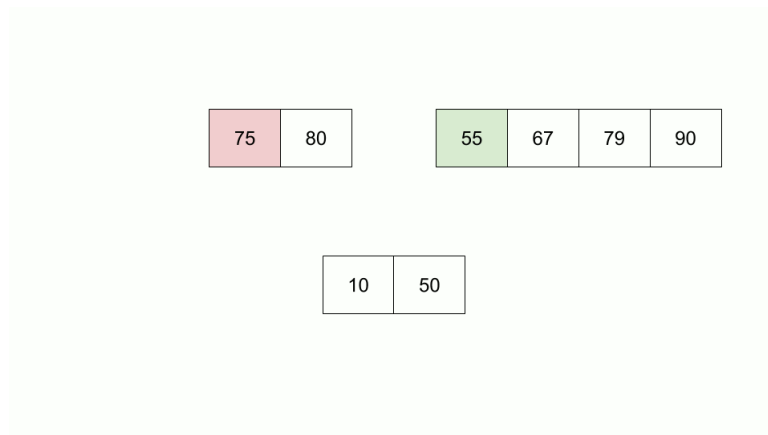


Figure 1.7: Comparando o próximo elemento da primeira lista

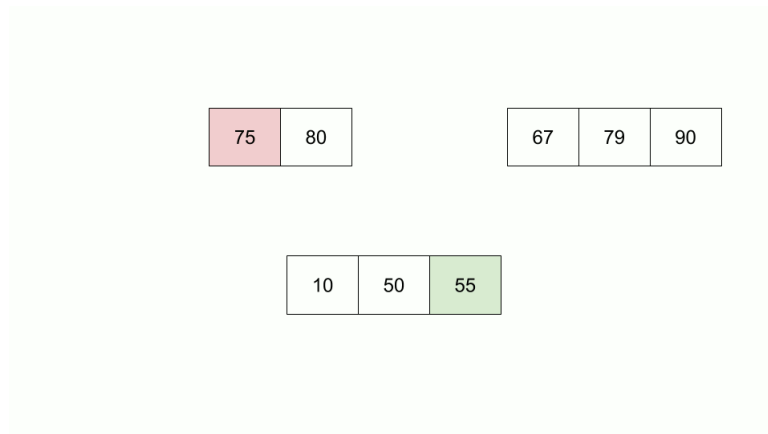


Figure 1.8: 55 é menor, então é anexado no final da lista resultado

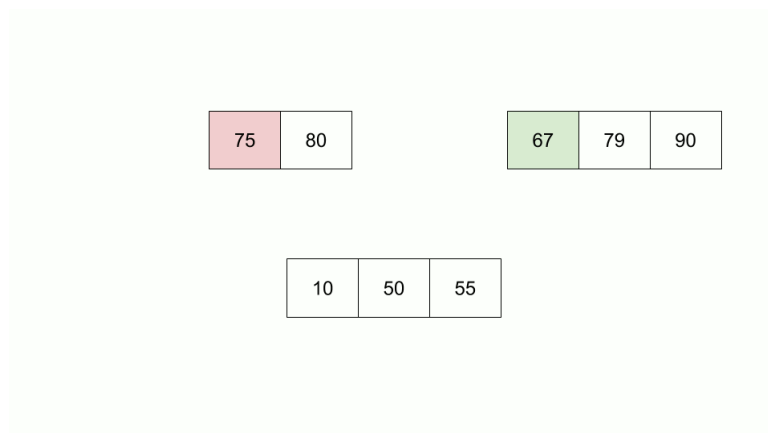


Figure 1.9: Comparando o próximo elemento da segunda lista

E o algoritmo vai continuar até que a lista resultado esteja completa com os elementos da primeira e segunda lista.

Dito isso, construímos o pseudocódigo dessa função da seguinte forma:

Input: $\text{listaEsquerda} = x_0, x_1, \dots, x_{N-1}$, $\text{listaDireita} = y_0, y_1, \dots, y_{M-1}$
Output: A **listaResultado** ordenada com os elementos da **listaEsquerda** e **listaDireita**

```

1: function Merge(listaEsquerda, listaDireita)
2:    $E, D, R = 0$                                 ▷ Esses serão os indexadores de cada lista
3:   listaResultado = 0, 0, ..., 0
4:   while  $E < N$  e  $D < M$  do
5:     if listaEsquerda( $E$ ) < listaDireita( $D$ ) then
6:       listaResultado( $R$ ).push(listaDireita( $D$ ))
7:        $E = E + 1$                                 ▷ Partimos para o próximo elemento
8:     else
9:       listaResultado( $R$ ).push(listaEsquerda( $D$ ))
10:       $D = D + 1$ 
11:    end if
12:     $R = R + 1$ 
13:  end while
14:  ▷ Como uma das listas vai esgotar primeiro que a outra, copiamos os elementos
15:  restantes para a listaResultado
16:  while listaEsquerda ou listaDireita tiver elementos do
17:    adicione os elementos na listaResultado
18:  end while
19:  return listaResultado
20: end function

```

1.2.4 Análise de complexidade

Para analisar qual é a complexidade de tempo da **Merge sort**, nas suas duas versões, vamos estabelecer primeiro qual é a complexidade da função **Merge**. A partir do [pseudocódigo](#) da função, é fácil de ver que durante sua execução ela sempre vai percorrer o tamanho máximo entre a **listaEsquerda** e **listaDireita**, portanto, sua complexidade na notação de complexidade assintótica é $O(n)$, $\Theta(n)$ e $\Omega(n)$.

Análise da versão iterativa

TODO

Análise da versão recursiva

Analisando o corpo da função, teremos dois casos: caso o tamanho da lista (n) seja menor ou igual 1 e caso contrário.

```

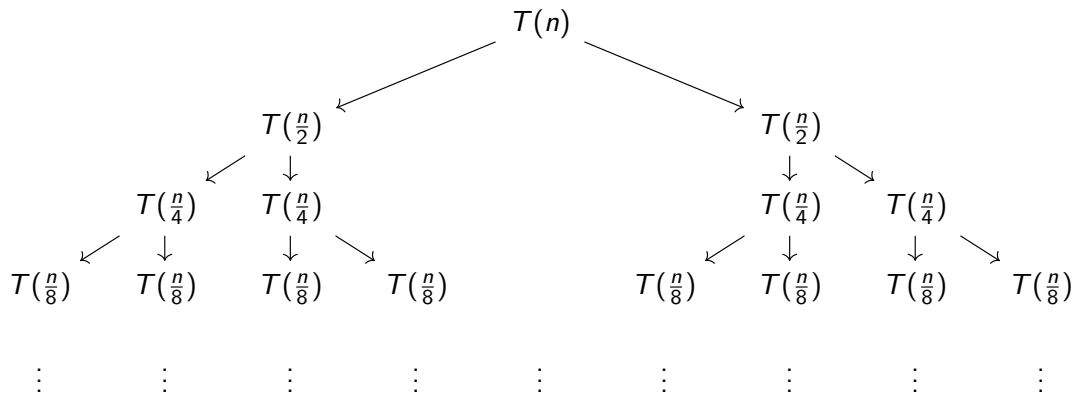
if tamanho da lista  $\leq 1$  then return
end if
return Merge(MergeSort(1ª metade da lista), MergeSort(2ª metade da lista))

```

No primeiro caso, é fácil de ver que a complexidade da função para uma lista de qualquer tamanho é constante, ou seja, $O(1)$, $\Theta(1)$ e $\Omega(1)$. Caso contrário, observa-se que a função chama a si mesma duas vezes, uma para cada metade da lista. Ademais, a função *Merge*, com complexidade linear para qualquer entrada, é chamada em cada etapa recursiva. Portanto, estabelecemos a relação de recorrência da *Merge sort* recursiva como:

$$T(n) = \begin{cases} O(1), & \text{se } n \leq 1 \\ 2T(\frac{n}{2}) + O(n), & \text{caso contrário} \end{cases}$$

Uma vez estabelecida a relação de recorrência, vamos usar primeiro o método da **Árvore de recorrência**. Começemos estabelecendo seu diagrama:



Uma vez estabelecida a árvore de recorrência, podemos tabular o tamanho da entrada, seu custo por nó e quantidade de nós para cada nível da árvore:

Nível da árvore	Tamanho da entrada	Custo por nó	Quantidade de nós
0	n	n	$1 = 2^0$
1	$\frac{n}{2^1}$	$\frac{n}{2^1}$	$2 = 2^1$
2	$\frac{n}{2^2}$	$\frac{n}{2^2}$	$4 = 2^2$
3	$\frac{n}{2^3}$	$\frac{n}{2^3}$	$8 = 2^3$
\vdots	\vdots	\vdots	\vdots
i	$\frac{n}{2^i}$	$\frac{n}{2^i}$	2^i

Em seguida, para estabelecer o somatório que calcula a complexidade da função, precisamos identificar o valor de i para quando $T(\frac{n}{2^i}) = T(1)$, assim, teremos:

$$\begin{aligned} \frac{n}{2^i} = 1 &\implies n = 2^i \\ &\implies \log_2 n = i \end{aligned}$$

Dessa forma, a complexidade da função para Ω , Θ e O será dada pelo resultado do somatório:

$$\begin{aligned}
\sum_{i=0}^{\log_2 n} \frac{n}{2^i} \cdot 2^i &= \sum_{i=0}^{\log_2 n} n \\
&= n \sum_{i=0}^{\log_2 n} 1 \\
&= n \cdot \log_2 n
\end{aligned}$$

Pelo método do **Teorema mestre**, o mesmo resultado ocorre em ainda menos etapas. Pelo teorema, temos que estabelecendo a relação de recorrência nessa forma:

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^k)$$

Para algum $a \geq 1$, $b \geq 1$, e $k \geq 0$. Vale que:

1. Se $a \geq b^k$, então $T(n)$ é $\Theta(n^{\log_b a})$.
2. Se $a = b^k$, então $T(n)$ é $\Theta(n^k \cdot \log_b a)$.
3. Se $a < b^k$, então $T(n)$ é $\Theta(n^k)$.

A partir da [relação de recorrência estabelecida](#), tome $a = 2$, $b = 2$ e $k = 1$. Como $b^k = 2 = a$, logo, $T(n) = \Theta(n \cdot \log_2 n)$. Como foi estabelecido que o pior caso, o melhor e caso médio iam ter a mesma complexidade, logo, a *Merge sort* também tem as complexidades $O(n \cdot \log_2 n)$ e $\Omega(n \cdot \log_2 n)$.

1.3 Quick Sort

1.4 Quick Sort

Assim como o Merge Sort, o Quick Sort é um algoritmo baseado na técnica de divisão e conquista. A operação ocorre da seguinte forma:

1. **Dividir:** O vetor $A[p..r]$ é dividido em dois subvetores não vazios $A[p..q]$ e $A[q+1..r]$. O índice q é escolhido a partir do elemento localizado na metade do vetor original, denominado pivô. Os elementos do vetor são rearranjados de modo que os elementos à esquerda de q sejam menores ou iguais ao pivô, e os elementos à direita sejam maiores ou iguais ao pivô.
2. **Conquistar:** Os dois subvetores $A[p..q]$ e $A[q+1..r]$ são ordenados através de chamadas recursivas ao Quick Sort.
3. **Combinar:** Esta etapa não exige nenhum processamento adicional, pois, ao longo do processo recursivo, os elementos vão sendo ordenados no próprio vetor.

1.4.1 Quick Sort Iterativo

1.4.2 Quick Sort Recursivo

Algorithm 3 Iterative Quick Sort

Input: Lista A **Output:** Lista A ordenada

```

1: function particao( $A, p, r$ )
2:   declare  $pivo, i, j$  como numérico
3:    $pivo \leftarrow A[(p + r)/2]$ 
4:    $i \leftarrow p - 1$ 
5:    $j \leftarrow r + 1$ 
6:   while  $i < j$  do
7:     repeat
8:        $j \leftarrow j - 1$ 
9:     until  $A[j] \leq pivo$ 
10:    repeat
11:       $i \leftarrow i + 1$ 
12:    until  $A[i] \geq pivo$ 
13:    if  $i < j$  then
14:      troca( $A, i, j$ )
15:    end if
16:  end while
17:  return  $j$ 
18: end function

19: function IterativeQuickSort( $A$ )
20:   if  $A$  estiver vazio then
21:     return
22:   end if
23:   Crie uma pilha com o par  $(0, \text{tamanho de } A - 1)$ 
24:   while a pilha não estiver vazia do
25:     Remova  $(low, high)$  do topo da pilha
26:     if  $low < high$  then
27:        $pivot\_index \leftarrow \text{Particao}(A, low, high)$ 
28:       if  $pivot\_index > 0$  then
29:         Adicione  $(low, pivot\_index - 1)$  na pilha           ▷ Lado esquerdo
30:       end if
31:       Adicione  $(pivot\_index + 1, high)$  na pilha           ▷ Lado direito
32:     end if
33:   end while
34: end function

```

Algorithm 4 Quick Sort

Input: Lista $A = A_1, A_2, \dots, A_n$ **Output:** Lista A ordenada

```
1: function troca( $A, i, j$ )
2:   declare  $aux$  como numérico
3:    $aux \leftarrow A[i]$ 
4:    $A[i] \leftarrow A[j]$ 
5:    $A[j] \leftarrow aux$ 
6: end function

7: function particao( $A, p, r$ )
8:   declare  $pivo, i, j$  como numérico
9:    $pivo \leftarrow A[(p + r)/2]$ 
10:   $i \leftarrow p - 1$ 
11:   $j \leftarrow r + 1$ 
12:  while  $i < j$  do
13:    repeat
14:       $j \leftarrow j - 1$ 
15:    until  $A[j] \leq pivo$ 
16:    repeat
17:       $i \leftarrow i + 1$ 
18:    until  $A[i] \geq pivo$ 
19:    if  $i < j$  then
20:      troca( $A, i, j$ )
21:    end if
22:  end while
23:  return  $j$ 
24: end function

25: function quickSort( $A, p, r$ )
26:   declare  $q$  como numérico
27:   if  $p < r$  then
28:      $q \leftarrow$  particao( $A, p, r$ )
29:     quickSort( $A, p, q$ )
30:     quickSort( $A, q + 1, r$ )
31:   end if
32: end function
```

Análise do Quick Sort Recursivo

Primeiro, devemos definir a relação de recorrência do algoritmo.

No procedimento de partição, o tempo de execução é limitado pelo tamanho n do vetor. Isso ocorre porque ele compara todos os elementos do vetor com o pivô enquanto os índices atenderem a condição $i < j$. Logo, o procedimento de partição realizará $O(n)$ comparações.

Os dois vetores gerados pelo procedimento de partição são resolvidos recursivamente. O tamanho desses vetores depende do valor do pivô escolhido na função de partição. Suponha que k elementos estejam ao lado esquerdo do pivô e $(n - k - 1)$ elementos estejam à direita do pivô após a partição.

Logo, a complexidade do passo recursivo será a soma das recorrências da ordenação dos dois vetores: $T(k) + T(n - k - 1)$.

Somando a parte recursiva do algoritmo com o procedimento de partição, teremos:

$$T(n) = O(n) + T(k) + T(n - k - 1)$$

O tempo de execução do Quick Sort depende se o particionamento é ou não balanceado. Se for balanceado, o algoritmo executa tão rapidamente quanto o Merge Sort; caso contrário, ele executará tão lentamente quanto o Insertion Sort. Assim, temos dois casos:

- **Pior caso:** Ocorre quando o pivô é o maior ou o menor elemento do vetor. Aqui, um vetor terá $n - 1$ elementos e o outro vetor será vazio (não se esqueça do pivô).

Para calcular a complexidade do Quick Sort no pior caso, substituímos $k = n - 1$ na recorrência encontrada:

$$\begin{aligned} T(n) &= O(n) + T(n - 1) + T(n - n + 1 - 1) \\ &= cn + T(n - 1) \quad (\text{considere } c \text{ uma constante}) \end{aligned}$$

- **Melhor caso:** Ocorre quando o pivô é o elemento médio do vetor a ser ordenado em cada chamada do algoritmo de partição. Nessa situação, o processo de partição será balanceado e o tamanho de cada vetor gerado pela partição será, aproximadamente, $n/2$.

Para calcular a complexidade do Quick Sort nesse caso, substituímos $k = n/2$ na recorrência encontrada:

$$\begin{aligned} T(n) &= O(n) + T(n/2) + T(n - n/2 - 1) \\ &\approx cn + 2T(n/2) \end{aligned}$$

1. Análise do melhor caso:

- (a) **Substituição:** Aqui, mostraremos, por indução, que $T(n) = cn + 2T(n/2)$ é limitada por $f(n) = n^2$, para todo $n \geq 2$.

- **Caso base:** $n = 2$.

Sabemos que $T(2) = 2c + 2T(1) = 4$ e $f(2) = 4$.

Logo, $T(2) \leq f(2)$.

Portanto, a base da indução é válida.

- **Hipótese de Indução:** Suponha que $T(k) \leq k^2$ para todo $2 < k \leq n$.

- **Passo indutivo:** Queremos provar que $T(k+1) \leq (k+1)^2$.
Observe que:

$$T(n+1) = cn + 2T\left(\frac{n+1}{2}\right)$$

(b) **Iteração:**

(c) **Método Mestre:** Esse método permite resolver recorrências da forma

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^k)$$

com $a \geq 1$, $b > 1$, e $k \geq 0$, onde a , b , e k são constantes.

Para a recorrência $T(n) = cn + 2T\left(\frac{n}{2}\right)$, temos que $a = 2$, $b = 2$ e $f(n) = n$.

Além disso, sabemos que $\log_b a = \log_2 2 = 1$.

Como $f(n) = n = O(n^{\log_2 2}) = O(n)$, então $T(n) = \Theta(n \log n)$.

(d) **Árvore Binária:**

Capítulo 2

Análise de Algoritmo

2.1 Função iterativa

2.2 Função recursiva

2.3 Implementação dos algoritmos de ordenação

Os algoritmos de ordenação **Merge Sort**, **Quick Sort** e **Bubble Sort** foram implementados em suas versões iterativas e recursivas na linguagem de programação Rust. Além disso, com a exceção do **Bubble sort**, todas as funções esperam um slice mutável de tipo T que implementa uma ordem parcial e pode ser copiado, estes são os traits `PartialOrd` e `Copy`, respectivamente. Todas as funções alteram esse slice diretamente e não retornam nada.

2.3.1 Bubble Sort

Iterativo

```
1 pub fn iterative_bubble_sort<T: PartialOrd>(arr: &mut [T]) {
2     for i in 0..arr.len() {
3         for j in 1..arr.len() - i {
4             if arr[j - 1] > arr[j] {
5                 arr.swap(j - 1, j);
6             }
7         }
8     }
9 }
```

Recursivo

Na versão recursiva, foi utilizada a função auxiliar `bubble_sort_pass`

```
1 pub fn recursive_bubble_sort<T: PartialOrd>(arr: &mut [T]) {
2     if arr.is_empty() {
3         return;
4     }
5     let last_element_position = arr.len();
6     bubble_sort_pass(arr, 1, last_element_position);
7     recursive_bubble_sort(&mut arr[..last_element_position - 1]);
8 }
```

Função auxiliar

Essa função move o maior elemento de um slice para a posição `last_element_position` no `arr`. O `iterator` é usado para percorrer o slice do início ao fim.

```
1 fn bubble_sort_pass<T: PartialOrd>(arr: &mut [T], iterator: usize,
2   last_element_position: usize) {
3     if iterator >= last_element_position {
4       return;
5     }
6     if arr[iterator - 1] > arr[iterator] {
7       arr.swap(iterator - 1, iterator)
8     }
9     bubble_sort_pass(arr, iterator + 1, last_element_position)
10 }
```

2.3.2 Quick Sort

Iterativo

Como o *quick sort* naturalmente é um algoritmo recursivo, precisamos de algo que simulasse a pilha de execução do computador, esta simulação se deu a partir do uso da variável `stack` que armazena as posições das sub-listas que precisam ser ordenadas, permitindo a progressão iterativa da divisão da lógica de divisão e conquista.

```
1 pub fn iterative_quick_sort<T: PartialOrd + Copy>(arr: &mut [T]) {
2     if arr.is_empty() {
3       return;
4     }
5     let mut stack = vec![(0, arr.len() - 1)];
6     while let Some((low, high)) = stack.pop() {
7       if low < high {
8         let pivot_index = partition(arr, low, high);
9         if pivot_index > 0 {
10          stack.push((low, pivot_index - 1)); // Left side
11        }
12        stack.push((pivot_index + 1, high)); // Right side
13      }
14    }
15 }
```

Recursivo

Para manter o único parâmetro `arr`, a `recursive_quick_sort` só encapsula a versão de fato recursiva.

```
1 pub fn recursive_quick_sort<T: PartialOrd + Copy>(arr: &mut [T]) {
2     if arr.is_empty() {
3       return;
4     }
5     _recursive_quick_sort(arr, 0, arr.len() - 1);
6 }
```

```

1 fn _recursive_quick_sort<T: PartialOrd + Copy>(
2     arr: &mut [T],
3     lower_bound: usize,
4     upper_bound: usize,
5 ) {
6     if lower_bound >= upper_bound {
7         return;
8     }
9     let pivot_index = partition(arr, lower_bound, upper_bound);
10    if pivot_index > 0 {
11        _recursive_quick_sort(arr, lower_bound, pivot_index - 1);
12    }
13    _recursive_quick_sort(arr, pivot_index + 1, upper_bound);
14 }

```

Listing 2.1: Versão correta

Função auxiliar

Note que ambas as versões do *quick sort* utilizam a função auxiliar *partition*. Essa função seleciona o último elemento como pivô e rearranja a lista, de modo que todos os elementos menores que o pivô estejam na esquerda e os maiores ou iguais ao pivô na direita. Ao final, ela retorna o índice do pivô após a partição. Os parâmetros *lower_bound*, *upper_bound* são os índices do elemento inicial e final do slice a ser particionado.

```

1 fn partition<T: PartialOrd + Copy>(arr: &mut [T], lower_bound: usize,
2     upper_bound: usize) -> usize {
3     let pivot = arr[upper_bound];
4     let mut left_item = lower_bound as isize - 1;
5     for right_item in lower_bound..upper_bound {
6         if arr[right_item] < pivot {
7             left_item += 1;
8             arr.swap(left_item as usize, right_item);
9         }
10    }
11    let new_pivot = (left_item + 1) as usize;
12    arr.swap(new_pivot, upper_bound);
13    new_pivot
14 }

```

2.3.3 Merge Sort

Iterativo

Novamente, como o *merge sort* naturalmente é um algoritmo recursivo, foi necessário simular a pilha de execução do computador. Primeiro, começamos com pequenos segmentos da lista (de tamanho 1) e iterativamente dobramos o tamanho dos segmentos a cada passo, mesclando-os.

```

1 pub fn iterative_merge_sort<T: PartialOrd + Copy>(arr: &mut [T]) {
2     if arr.len() <= 1 {
3         return;
4     }
5     let mut temp_arr = arr.to_vec();
6     let mut segment_size = 1;
7     let arr_len = arr.len();
8     while segment_size < arr_len {
9         let mut start = 0;

```

```

10     while start < arr_len {
11         let mid = (start + segment_size).min(arr_len);
12         let end = (start + 2 * segment_size).min(arr_len);
13         merge(&mut temp_arr[start..end], &arr[start..mid], &arr[mid..end]);
14         start += 2 * segment_size;
15     }
16     arr.copy_from_slice(&temp_arr);
17     segment_size *= 2;
18 }
19 }

```

Recursivo

A principal diferença em relação [pseudo código](#), é que a agora a função edita diretamente a lista, em vez de retorna-la como resultado.

```

1 pub fn recursive_merge_sort<T: PartialOrd + Copy>(arr: &mut [T]) {
2     if arr.len() <= 1 {
3         return;
4     }
5     let mid = arr.len() / 2;
6     let mut left_arr = arr[..mid].to_vec();
7     let mut right_arr = arr[mid..].to_vec();
8     recursive_merge_sort(&mut left_arr);
9     recursive_merge_sort(&mut right_arr);
10    merge(arr, &left_arr, &right_arr);
11 }

```

Função auxiliar

E a *merge* também foi alterada de forma parecida, de forma que receba como argumento a lista resultado (arr).

```

1 fn merge<T: PartialOrd + Copy>(arr: &mut [T], left_arr: &[T], right_arr:
    &[T]) {
2     let left_arr_len = left_arr.len();
3     let right_arr_len = right_arr.len();
4     let (mut i, mut l, mut r) = (0, 0, 0);
5     while l < left_arr_len && r < right_arr_len {
6         if left_arr[l] < right_arr[r] {
7             arr[i] = left_arr[l];
8             l += 1;
9         } else {
10            arr[i] = right_arr[r];
11            r += 1;
12        }
13        i += 1;
14    }
15    while l < left_arr_len {
16        arr[i] = left_arr[l];
17        i += 1;
18        l += 1;
19    }
20    while r < right_arr_len {
21        arr[i] = right_arr[r];
22        i += 1;
23        r += 1;
24    }
25 }

```

2.3.4 Organização do projeto e corretude do algoritmo

Para testar os algoritmos o projeto foi subdividido em duas partes, a que testa a performance e a que estipula a corretude, para tal foi usado a separação padrão do cargo entre o módulo principal e o módulo de testes. A organização dos arquivos ocorreu da seguinte forma:

```

/
├── out/
│   ├── entries.txt
│   └── output.txt
├── src/
│   ├── algorithms.rs
│   ├── lib.rs
│   └── main.rs
├── tests/
│   └── test_sorts.rs
├── Cargo.lock
├── Cargo.toml
└── rustfmt.toml

```

No Cargo.toml A unica dependência usada foi `rand`, para poder gerar números aleatórios de 0 a 100000

```

1  [dependencies]
2  rand = "0.8.5"

```

Listing 2.2: Trecho do Cargo.toml

Quando executado com cargo test, o cargo executa as funções de teste definidas em test_sorts.rs para estipular a corretude de cada algoritmo. Os testes definidos foram:

```

1  fn reverse_list_test(func: fn(&mut [i32])) {
2      let mut arr = [5, 3, 2, 4, 1];
3      func(&mut arr);
4      assert_eq!(arr, [1, 2, 3, 4, 5], "reverse_list_test failed");
5  }
6
7  fn duplicates_list_test(func: fn(&mut [i32])) {
8      let mut arr = [4, 2, 3, 2, 1, 4];
9      func(&mut arr);
10     assert_eq!(arr, [1, 2, 2, 3, 4, 4], "duplicates_list_test failed");
11 }
12
13 fn already_sorted_list_test(func: fn(&mut [i32])) {
14     let mut arr = [1, 2, 3, 4, 5];
15     func(&mut arr);
16     assert_eq!(arr, [1, 2, 3, 4, 5], "already_sorted_list_test failed");
17 }
18
19 fn singleton_list_test(func: fn(&mut [i32])) {
20     let mut arr = [42];
21     func(&mut arr);
22     assert_eq!(arr, [42], "singleton_list_test failed");
23 }
24
25 fn empty_list_test(func: fn(&mut [i32])) {
26     let mut arr: [i32; 0] = [];
27     func(&mut arr);

```



```

28  assert_eq!(arr, [], "empty_list_test failed");
29 }

```

Listing 2.3: Trecho de test_sorts.rs

Todos os testes unitários foram definidos como esse:

```

1  #[test]
2  fn test_recursive_quick_sort() {
3      reverse_list_test(recursive_quick_sort);
4      duplicates_list_test(recursive_quick_sort);
5      already_sorted_list_test(recursive_quick_sort);
6      singleton_list_test(recursive_quick_sort);
7      empty_list_test(recursive_quick_sort);
8  }

```

Listing 2.4: Trecho de test_sorts.rs

Executando o comando `cargo test`, podemos ver que todos os testes foram bem sucedidos:

```

1  running 6 tests
2  test test_iterative_bubble_sort ... ok
3  test test_iterative_merge_sort ... ok
4  test test_recursive_bubble_sort ... ok
5  test test_iterative_quick_sort ... ok
6  test test_recursive_merge_sort ... ok
7  test test_recursive_quick_sort ... ok
8
9  test result: ok. 6 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
   ; finished in 0.00s

```

Os testes de performance foram implementados em `main.rs`, onde criam e escrevem os arquivos `entries.txt` e `output.txt` com as listas geradas e os resultados de performance, respectivamente.

```

1  type SortFn<T> = fn(&mut [T]);
2
3  fn main() -> io::Result<()> {
4      let sort_functions = Vec::from([
5          ("ITE BUBBLE SORT", iterative_bubble_sort as SortFn<i32>),
6          ("REC BUBBLE SORT", recursive_bubble_sort),
7          ("REC QUICK SORT", recursive_quick_sort),
8          ("ITE QUICK SORT", iterative_quick_sort),
9          ("REC MERGE SORT", recursive_merge_sort),
10         ("ITE MERGE SORT", iterative_merge_sort),
11     ]);
12
13     fs::create_dir_all("out")?;
14     let mut entries_file = File::create(ENTRIES_FILENAME)?;
15     let mut output_file = File::create(OUTPUT_FILENAME)?;
16
17     let title = "Performance test for sort algorithms";
18     writeln!(output_file, "{}\n{}\n", title, "=" . repeat(title.len()))?;
19     writeln!(
20         output_file,
21         "ITE - stands for iterative\nREC - stands for recursive\n"
22     )?;
23
24     for n in 1..=4 {
25         writeln!(entries_file, "List with {} entries:\n", TEN.pow(n))?;
26         run_entry(
27             &sort_functions,
28             TEN.pow(n),

```

```

29     &mut entries_file,
30     &mut output_file,
31     )?;
32     writeln!(entries_file)?;
33 }
34
35 Ok(())
36 }

```

Listing 2.5: Trecho de main.rs

2.4 Análise de performance dos algoritmos de ordenação

Uma vez estabelecidas as implementações dos algoritmos de ordenação, em conjunto com testes que estipulavam sua corretude. Executamos o programa principal com todas as otimizações do compilador (`cargo run --release`) e obtivemos os seguintes resultados:

Table 2.1: Tabela de resultados

Algoritmo	10 entradas	100 entradas	1000 entradas	10000 entradas
Bubble Sort (Iterativo)	281 ns	6.723 µs	314.678 µs	31.196169 ms
Bubble Sort (Recursivo)	221 ns	7.374 µs	411.449 µs	43.688736 ms
Quick Sort (Recursivo)	320 ns	2.554 µs	31.329 µs	388.807 µs
Quick Sort (Iterativo)	581 ns	2.775 µs	33.122 µs	403.624 µs
Merge Sort (Recursivo)	882 ns	5.69 µs	62.497 µs	744.751 µs
Merge Sort (Iterativo)	531 ns	2.976 µs	36.809 µs	467.994 µs

Começando pelo **Bubble Sort**, é possível notar que, apesar de uma performance inicial superior da versão recursiva, a implementação iterativa é mais eficiente conforme o tamanho da lista aumenta. Acredita-se que o overhead causado pelas chamadas recursivas e pelo contexto adicional na pilha de execução (função auxiliar recursiva) resultou em uma performance pior em comparação com a iterativa.

Em seguida, no **Quick Sort**, tivemos um cenário diferente, em que a versão recursiva se saiu como a mais eficiente em todas as entradas em relação à iterativa. Isso pode ter ocorrido pela instanciamento da estrutura de dados pilha como um `Vec<(usize, usize)>`, que fica armazenado na heap, e um overhead de operações `push` e `pop`, enquanto as chamadas recursivas não lidam com essas operações e ficam diretamente na pilha do sistema, que é mais rápida.

Por último, no **Merge Sort**, que, apesar de naturalmente ser um algoritmo recursivo, ficou muito atrás de sua versão iterativa, a qual foi até duas vezes mais rápida na maioria dos casos. Entretanto, é fácil entender a origem dessa diferença, considerando que, na implementação recursiva, são instanciados dois `Vec<T>` em cada chamada recursiva, enquanto na versão iterativa é instanciado um único `Vec<T>` durante toda a execução.

Referências Bibliográficas

Contribuidores da Wikipedia (2024a), 'Divide-and-conquer algorithm — wikipedia, the free encyclopedia'. [Acessado: 26 de Outubro, 2024].

URL: https://en.wikipedia.org/wiki/Divide-and-conquer_algorithm

Contribuidores da Wikipedia (2024b), 'John von neumann — wikipedia, the free encyclopedia'. [Acessado: 26 de Outubro, 2024].

URL: https://pt.wikipedia.org/wiki/John_von_Neumann

Contribuidores da Wikipedia (2024c), 'Merge sort'. [Acessado: 26 de Outubro, 2024].

URL: https://en.wikipedia.org/wiki/Merge_sort

Contribuidores da Wikipedia (2024d), 'Quicksort — wikipedia, the free encyclopedia'. [Acessado: 26 de Outubro, 2024].

URL: <https://en.wikipedia.org/wiki/Quicksort>

Contribuidores do GeeksforGeeks (2024a), 'Merge sort'. [Acessado: 26 de Outubro, 2024].

URL: <https://www.geeksforgeeks.org/merge-sort/>

Contribuidores do GeeksforGeeks (2024b), 'Quick sort'. [Acessado: 26 de Outubro, 2024].

URL: <https://www.geeksforgeeks.org/quick-sort-algorithm/>

Mahmud, S. (2024), 'Merge sort: A detailed explanation'. [Acessado: 26 de Outubro, 2024].

URL: <https://blog.shahadmahmud.com/merge-sort/>