Universidade Federal do Rio Grande do Norte
Instituto Metrópole Digital

Análise de algoritmos clássicos de ordenação e busca

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

Estrutura de Dados Básica II

Trabalho da Primeira Unidade

7 de novembro 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 . . . 2

1.1.3 Análise de complexidade . . . 2

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 . . . 11

1.3 Quick Sort . . . 16

1.3.1 Quick Sort Recursivo . . . 17

1.3.2 Quick Sort Iterativo . . . 17

1.3.3 Função auxiliar Partição . . . 18

1.3.4 Análise de Complexidade . . . 18

2 Análise de Algoritmo **21**

2.1 Ambiente computacional . . . 21

2.2 Função iterativa . . . 21

2.2.1 Algoritmo idadeRep . . . 21

2.2.2 Análise do idadeRep . . . 22

2.2.3 Resultados do idadeRep . . . 22

2.2.4 Algoritmo idadeRep2 . . . 23

2.2.5 Análise do idadeRep2 . . . 23

2.2.6 Resultados do idadeRep2 . . . 23

2.2.7 Comparação dos algoritmos . . . 23

2.3 Função Recursiva . . . 24

2.3.1 Algoritmo buscaBinaria . . . 24

2.3.2 Análise do buscaBinaria . . . 24

2.3.3 Resultados do buscaBinaria . . . 25

2.3.4 Algoritmo bBinRec . . . 25

2.3.5 Análise do bBinRec . . . 25

2.3.6 Resultados do bBinRec . . . 26

2.3.7 Comparação dos algoritmos . . . 26

2.4 Implementação dos algoritmos de ordenação . . . 26

2.4.1 Bubble Sort . . . 27

2.4.2 Quick Sort . . . 27

2.4.3 Merge Sort . . . 29

2.4.4 Organização do projeto e corretude do algoritmo . . . 31

2.5 Análise de performance dos algoritmos de ordenação 33

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++)
```

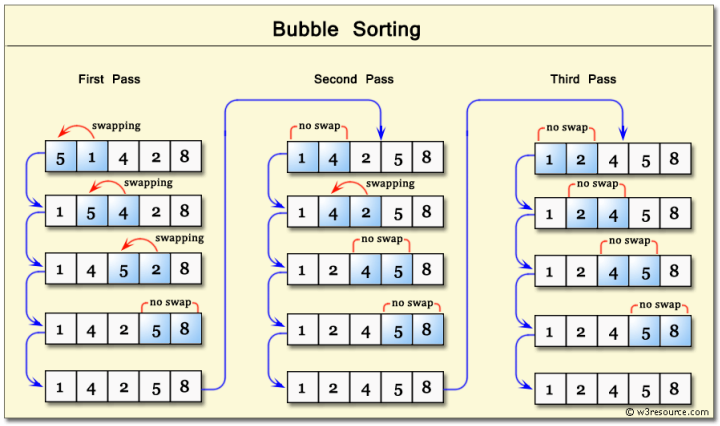


Figure 1.1: Diagrama exemplo de um Bubble Sort

1.1.1 Bubble Sort Iterativo

Dessa forma, estabelecemos o pseudocódigo de sua versão iterativa como:

Algorithm 1 Bubble Sort

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

```
1: function BubbleSort(A)
2:     for j ← 1 to n − 1 do
3:         for i ← 0 to n − 2 do
4:             if A[i] > A[j + 1] then
5:                 troque i por j em A
6:             end if
7:         end for
8:     end for
9: end function
```

1.1.2 Bubble Sort Recursivo

Para sua versão recursiva, vamos analisar o algoritmo em dois casos. Quando uma lista tem no máximo 1 elemento, ela está ordenada, então esse será o caso base. Para o caso recursivo, pelo teorema da recursão podemos assumir que a função já "borbulhou"todos os $n-1$ maiores elementos da lista para o final. Então, basta colocar o primeiro elemento em sua posição:

Algorithm 2 Bubble Sort Recursivo

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

```
1: function BubbleSortRecursivo(A, n)
2:     if n ≤ 1 then return
3:     end if
4:     for j ← 0 to n − 1 do
5:         if A[j] > A[j + 1] then
6:             troque i por j em A
7:         end if
8:     end for
9:     BubbleSortRecursivo(A, n − 1)
10: end function
```

1.1.3 Análise de complexidade

Análise da versão iterativa

No Bubble Sort, 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:

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

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.

Análise da versão recursiva

Note que assim como na versão iterativa do Bubble Sort, a função percorre a lista a ser ordenada n vezes e logo é chamada novamente para o tamanho $n-1$. Como haverá $n-1$ chamadas até que chegue no caso base, em que $n \leq 1$, estabelecemos a relação de recorrência do Bubble Sort como:

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

Dada a relação de recorrência desse algoritmo, vamos analisar sua complexidade com o 4 métodos de análise estudados: **Substuição, Iteração, Árvore de Recurso e Teorema Mestre**.

1. **Subsituição**:

   Nesse método, mostraremos, por indução, que $T(n) = T(n-1) + n$ é limitada por $f(n) = n^2$, ou seja $T(n) \leq n^2$. Portanto:

   - **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.
   - **Passo Indutivo:**
     Suponha que $T(k) \leq k^2$ para todo $1 < k \leq n$ (hipótese de indução).
     Queremos provar que $T(n+1) \leq (n+1)^2$. Logo, observe que:

     $$\begin{aligned}
     & T(n) \leq n^2 && \text{(pela hipótese de indução)} \\
     \implies & T(n) + n \leq n^2 + 2n && \text{(pois } n \leq 2n\text{)} \\
     \implies & T(n) + n \leq n^2 + 2n && \text{(pois } n \leq 2n\text{)} \\
     \implies & T(n) + n + 1 \leq n^2 + 2n + 1 \\
     \implies & T(n+1) \leq (n+1)^2
     \end{aligned}$$

   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. **Iteração**:

   Aqui, procuramos expandir a recorrência até que um padrão seja identificado:

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

Uma vez identificado o padrão, observa-se que 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:

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

3. **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.

4. **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:

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

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

$$\begin{array}{c} T(n) \\ \downarrow \\ T(n-1) \\ \downarrow \\ T(n-2) \\ \downarrow \\ T(n-3) \\ \downarrow \\ T(n-i) \end{array}$$

   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:

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

Portanto, temos:

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

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 sub-listas ordenadas até que exista apenas uma sub-lista, esta sub-lista 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.

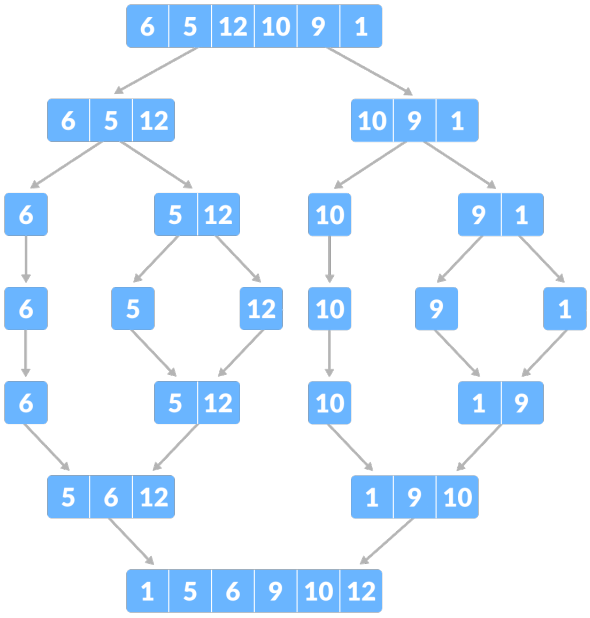


Figure 1.2: Diagrama exemplo de um Merge Sort

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 um 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: **lista** $= x_0, x_1, \ldots, x_{N-1}$

```
1: function MergeSort(lista)
2:     if tamanho da lista ≤ 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

E em contraste com a abordagem anterior, a forma iterativa do Merge Sort se aproveita de estruturas de repetição aninhadas para que tenha acesso a índices específicos. E tais índices quando usados como argumento na função merge fazem com que a lista "quebrada"seja reconstruída de maneira ordenada. De forma que o loop mais externo controle a quebra da lista em sub-listas com a metade do tamanho anterior e o loop mais interno controla o índice inicial de cada sub-lista, conforme o pseudocódigo abaixo.

Input: **lista** $= x_0, x_1, \ldots, x_{N-1}$

```
1: function MergeSortIt(lista)
2:     tamanhoListas ← 1            ▷ tamanho atual das sub-listas(variando de 1 até n/2)
3:     índiceEsq ← 1                ▷ index inicial da sub-lista à esquerda
4:     for tamanhoListas ≤ tamanho(lista) ; tamanhoListas = 2 · tamanhoListas do
5:         for índiceEsq < tamanho(lista) − 1 ; índiceEsq · = 2 · tamanhoListas do
6:             meio ← índiceEsquerda + tamanhoListas − 1
7:             fimDireita ← mínimo(índiceEsq + 2 · tamanhoListas, tamanho lista − 1)
8:             Merge(lista, índiceEsq, meio, fimDireita)
9:         end for
10:    end for
11: end function
```

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:

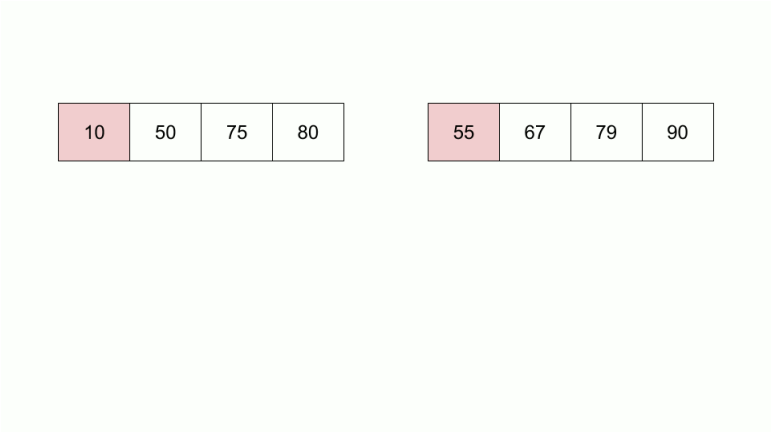


Figure 1.3: Comparando o primeiro elemento da primeira lista com o primeiro da segunda

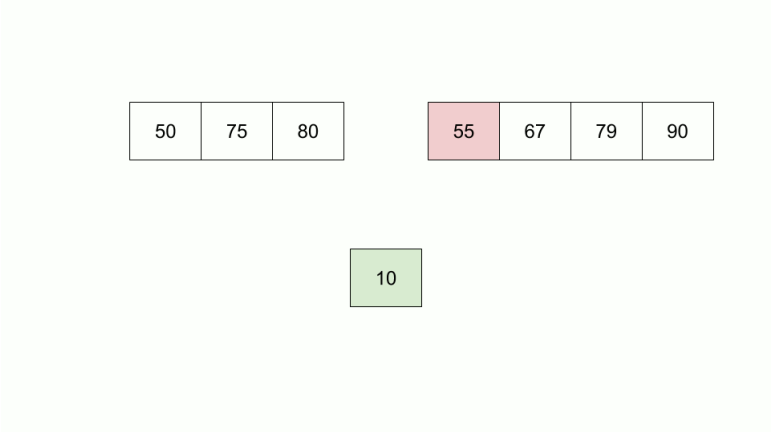


Figure 1.4: 10 é menor que 55, então é anexado no final da lista resultado

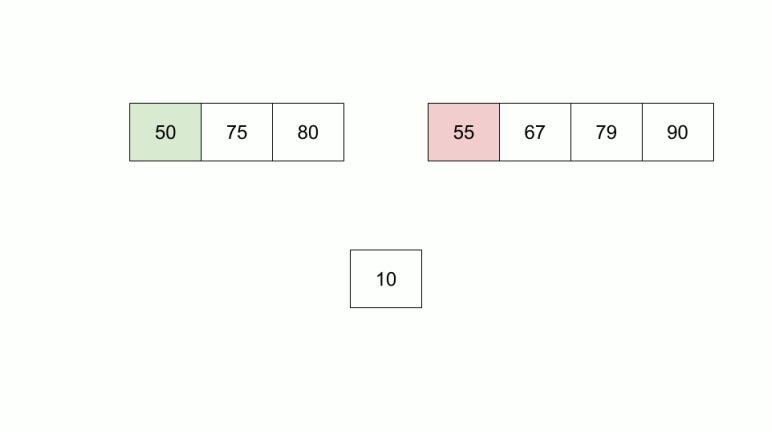


Figure 1.5: Comparando o próximo elemento da primeira lista

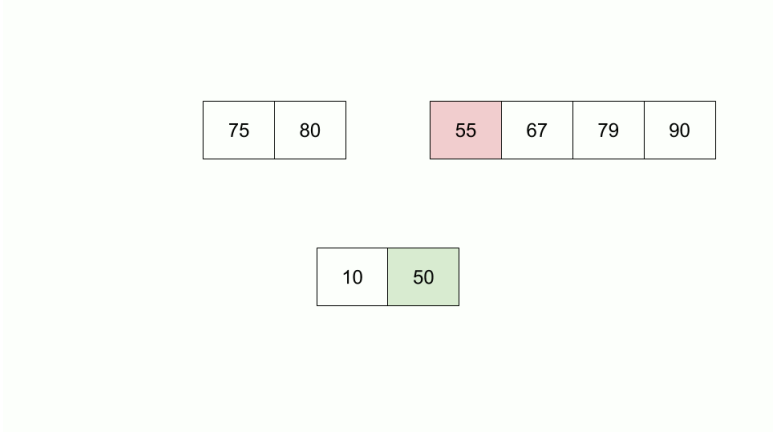


Figure 1.6: 50 é menor que 55, então é anexado no final da lista resultado

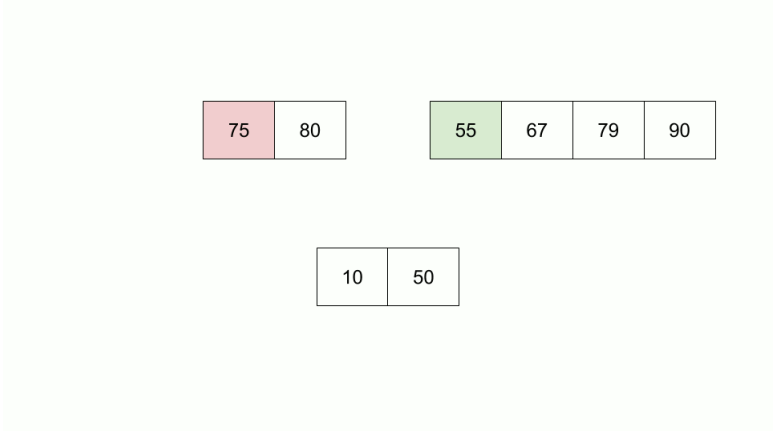


Figure 1.7: Comparando o próximo elemento da primeira lista

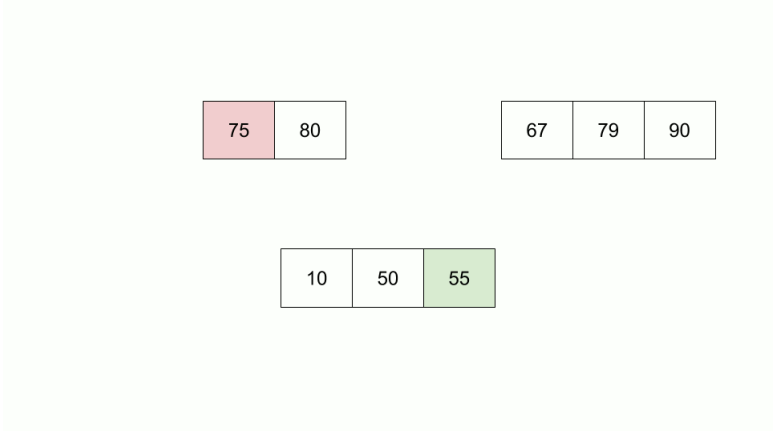


Figure 1.8: 55 é menor, então é anexado no final da lista resultado

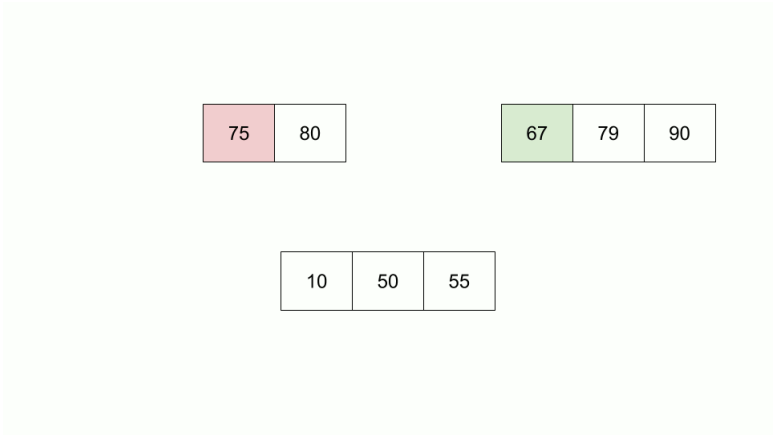


Figure 1.9: Comparando o próximo elemento da segunda lista

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: **listaEsquerda** $= x_0, x_1, \ldots, x_{N-1}$, **listaDireita** $= y_0, y_1, \ldots, y_{M-1}$
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 facil 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 $\Omega(n)$ para um tamanho de lista n.

Análise da versão iterativa

Observando a estrutura do pseudocódigo dessa versão é possível perceber que ela é estritamente dependente do aninhamento de loops, e nada mais. E simplificando o algoritmo podemos observá-lo da seguinte maneira:

```
for i ← 1; i ≤ n − 1; i = i · 2 do
    for j ← 0; j < n − 1; j+ = i · 2 do
        Merge()
    end for
end for
```

Além disso, a versão iterativa já começa com as sub-listas divididas em singletons a partir, restando apenas mesclar cada segmento e incrementar o seu tamanho.

```
tamanhoListas ← 1
while tamanhoListas ≤ tamanho(lista) ; tamanhoListas = 2· tamanhoListas do
    ...
end while
```

Sendo assim, apesar dos processos ocorrerem simultaneamente, podemos separar análise de complexidade da versão iterativa em duas partes: Reconstrução em sub-listas maiores e mesclagem ordenada das sub-listas.

Com a lista começando com **n** sub-listas unitárias, o primeiro passo é transformá-las em $\frac{n}{2}$ pares, e em seguida em $\frac{n}{4}$ quartetos, e depois em $\frac{n}{8}$ oitavos, até obtermos uma única lista de tamanho **n**.

Em cada iteração, o número de sublistas é dividido por 2, ou seja, o número de sublistas na iteração i é dado por $\frac{n}{2^i}$. O processo continua até que haja apenas uma sublista restante, o que ocorre quando:

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

Assim, o número total de iterações necessárias é $i = \log_2 n$. Como cada iteração envolve percorrer todas as sublistas para realizar as fusões, ou seja, a complexidade da função Merge, o custo de cada iteração é proporcional a **n**. Portanto, o tempo total de execução do algoritmo é:

$$\begin{aligned} T(n) &= O(n) \cdot O(\log_2 n) \\ &= O(n \cdot \log_2 n) \end{aligned}$$

Analisando da perspectiva do pior caso, perceba que mesmo se tratando de uma lista já ordenada ainda sim será necessário sub-dividí-la e reconstruí-la tal qual uma desordenada. Sendo assim, temos que a complexidade entre os cenários terá a mesma ordem independentemente do quão perto o algoritmos esteja do resultado desejado. Portanto, na notação assintótica a complexidade da Merge Sort iterativo é $O(n \cdot \log_2 n)$, $\Omega(n \cdot \log_2 n)$, $\Theta(n \cdot \log_2 n)$.

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 ≤ 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 mesmo 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, podemos analisar a complexidade da função utilizando os quatro métodos estudados no curso. Como o Merge Sort se comporta de maneira consistente em todos os casos—melhor, pior e médio—isso simplifica nossa análise. Optaremos por começar utilizando o método da **Substituição** para determinar a complexidade.

Primeiramente, queremos demonstrar que $T(n) \geq n \cdot \log_2 n$, a fim de encontrar a complexidade na notação Ω, que será equivalente às outras notações assintóticas. Por indução no n, teremos:

- Caso $n = 1$:

  Temos que:

  $$T(1) = 1 \ \& \ n \cdot \log_2 1 = 0$$

  Logo $T(1) \geq n \cdot \log_2 1$.

- Caso $n = k + 1$, para algum $k \in \mathbb{N}$:

  Suponha que $T(n) \geq n \cdot \log_2 n$ (hipótese de indução). Calculemos:

$$\begin{aligned} T(n+1) &= 2T\left(\frac{n+1}{2}\right) + O(n) \\ &\geq 2 \cdot \frac{n+1}{2} \cdot \log_2\left(\frac{n+1}{2}\right) + O(n) \qquad \text{(hipótese de indução e função crescente)} \\ &= (n+1)(\log_2(n+1) - 1) + O(n) \\ &= (n+1)\log_2(n+1)) - n + 1 + O(n) \\ &\geq (n+1)\log_2(n+1) \end{aligned}$$

  Dessa forma $T(n+1) \geq (n+1)\log_2(n+1)$.

Como também sabemos que o Merge Sort não excede $O(n \cdot \log_2 n)$ em termos de complexidade, podemos concluir que:

$$T(n) = \Theta(n \cdot \log_2 n)$$

Após analisar o Merge Sort pelo método da substituição, vamos dar continuidade ao nosso exercício ao analisá-lo pelo método da **Iteração**. Para tal, começaremos expandimos a recorrência até um padrão ser identificado:

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + n \\ &= 4T\left(\frac{n}{2}\right) + 2n \\ &= 8T\left(\frac{n}{2}\right) + 4n \\ &\vdots \\ &= 2^i T\left(\frac{n}{2^i}\right) + in \end{aligned}$$

Dessa forma, é identificado o caso base da relação a partir de:

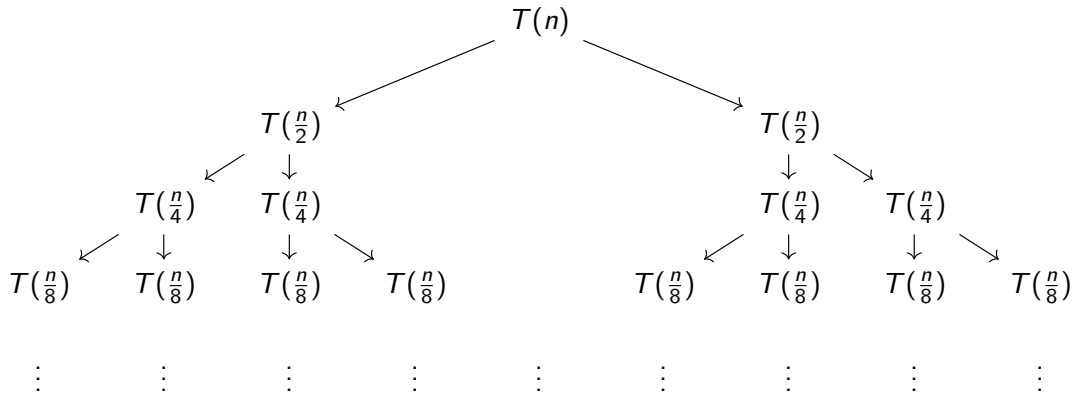
$$\begin{aligned}\frac{n}{2^i} = 1 &\implies n = 2^i \\ &\implies \log_2 n = i\end{aligned}$$

Substituindo, temos que:

$$\begin{aligned}&= n \cdot T(1) + n \cdot \log_2 n \\ &= n + n \cdot \log_2 n\end{aligned}$$

Portanto, a complexidade do Merge Sort se mantém consistente com o método da substituição e tem complexidade $n \cdot \log_2 n$ nas notações assintóticas.

Agora, vamos usar o método da **Árvore de recorrência** para verificar o mesmo resultado. Comecemos estabelecendo a árvore de recursão:



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	$\frac{n}{2^3}$	$\frac{n}{2^3}$	$8 = 2^3$
⋮	⋮	⋮	⋮
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:

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

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 $\Omega(n \cdot \log_2 n)$. E aqui terminamos essa análise.

1.3 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.

Para um melhor entendimento, segue uma esquema do primeiro laço de repetição realizado por este algoritmo:

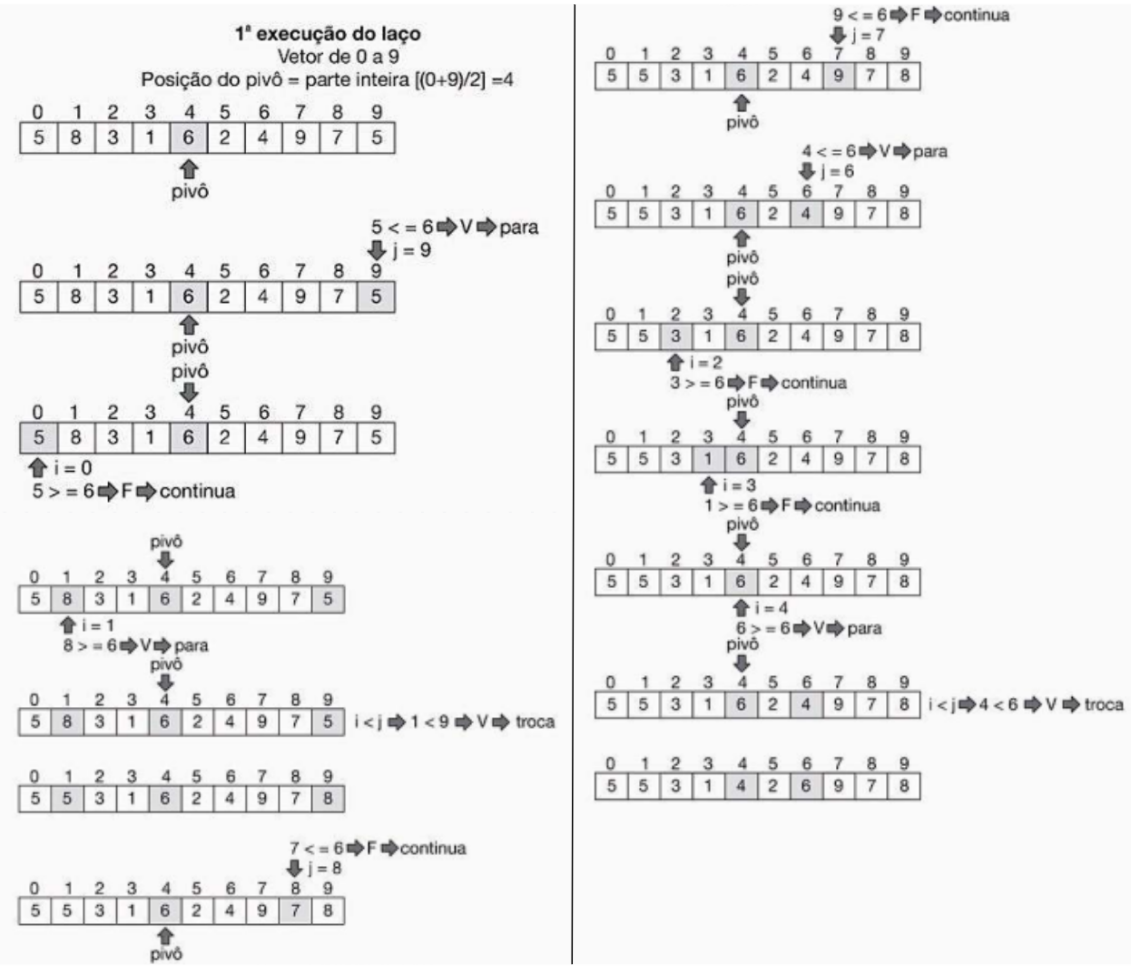


Figure 1.10: Disponível em: ASCÊNCIO, Ana Fernanda Gomes; ARAÚJO, Graziela Santos. Estruturas de Dados: Algoritmos, Análise da Complexidade e Implementações em Java e C/C++. Pearson; 1ª edição (30 setembro 2015)

1.3.1 Quick Sort Recursivo

Portanto, o caso base da Quick sort recursiva será o intervalo a ser ordenado for tiver 1 elemento ou for vazio. E a etapa recursiva será a chamada da função para a primeira metade até o pivô e segunda metade até o fim da lista.

Algorithm 3 Quick Sort Recursivo

```
1: function QuickSortRecursivo(A, limiteEsquerdo, limiteDireito)
2:     if limiteEsquerdo ≥ limiteDireito then return
3:     end if
4:     indicePivot ← particao(A, limiteEsquerdo, limiteDireito)
5:     quickSort(A, limiteEsquerdo, indicePivot)
6:     quickSort(A, q + 1, r)
7: end function
```

1.3.2 Quick Sort Iterativo

Dado que o Quick Sort é um algoritmo naturalmente recursivo, vamos emular a pilha de execução do computador usando a estrutura de dados pilha, cada chamada recursiva vai ser uma inserção na pilha, e cada etapa de execução será uma remoção na pilha. Com essas considerações, podemos copiar e colar o resto da implementação recursiva.

Algorithm 4 Iterative Quick Sort

Input: Lista A
Output: Lista A ordenada

```
 1: function IterativeQuickSort(A)
 2:     if A estiver vazio then
 3:         return
 4:     end if
 5:     Crie uma pilha com o par (0, tamanho de A − 1)
 6:     while a pilha não estiver vazia do
 7:         Remova (limiteEsquerdo, limiteDireito) do topo da pilha
 8:         if limiteEsquerdo ≥ limiteDireito then
 9:             continue
10:         end if
11:         Adicione (limiteEsquerdo, indicePivot − 1) na pilha      ▷ Lado esquerdo
12:         Adicione (indicePivot + 1, limiteDireito) na pilha       ▷ Lado direito
13:     end while
14: end function
```

1.3.3 Função auxiliar Partição

A função partição é responsável pela maior parte do trabalho braçal do Quick Sort, ela é quem organiza uma lista em relação a dterminado pivô

Algorithm 5 Partição

Input: A = $a_1, a_2, a_3, \ldots, a_n$, limiteEsquerdo, limiteDireito
Output: O índice do pivô

```
1:  function particao(A, limiteEsquerdo, limiteDireito)
2:      pivô ← A[limiteDireito]
3:      itemEsquerdo ← limiteEsquerdo
4:      for itemDireito ← limiteEsquerdo to limiteDireito do
5:          if A[itemDireito] < pivô then
6:              troque itemDireito por itemDireito em A
7:              itemEsquerdo ← itemEsquerdo +1
8:          end if
9:      end for
10:     troque itemEsquerdo por limiteDireito em A        ▷ O pivô é colocado no meio
11:     return itemEsquerdo
12: end function
```

1.3.4 Análise de Complexidade

Análise da versão recursiva

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:

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

$$= cn + T(n - 1) \quad \text{(considere } c \text{ uma constante)}$$

Observe que, neste caso, o algoritmo se comportará da mesma forma que o BubbleSort, que já foi analisado neste trabalho.

- **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:

$$T(n) = O(n) + T(n/2) + T(n - n/2 - 1)$$

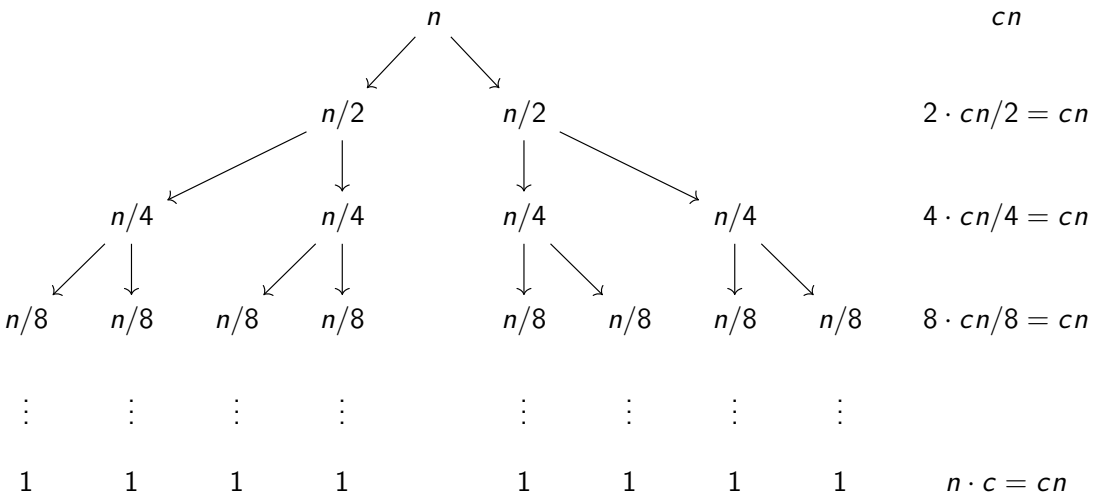
$$\approx cn + 2T(n/2)$$

  Agora, note que o Quick Sort, em seu melhor caso, se comporta da mesma forma do Merge Sort (1.2), dessa forma sua complexidade no melhor caso é $\Omega(n \cdot \log_2 n)$.

Análise da versão iterativa

Para determinar a complexidade da versão iterativa, precisamos estimar o tamanho da estrutura de dados "pilha"que simula as chamadas recursivas da outra versão. De forma que seja compreendido quando a estrutura de repetição "while"vai parar. Para tal, teremos dois casos:

- **Melhor caso**: Vamos supor que a função "partição"sempre particione a lista em duas sublistas iguais, ou seja, o pivô escolhido é sempre a mediana da lista maior, de forma que:



  Assim, a recorrência continuará até que $\frac{n}{2^i} = 1$, com i representando o nível da árvore representada acima. Com isso, teremos:

$$n = 2^i \implies i = \log_2 n$$

  Ou seja, a árvore terá $\log_2 n$ níveis, e cada nível da árvore de recursão realiza um trabalho proporcional a n, devido ao processo de particionamento. Portanto, a complexidade nesse caso será dada por $T(n) = n \log_2 n$.

  Assim, podemos dizer que o tempo de execução do algoritmo Quick Sort, em sua forma iterativa no melhor caso, será $O(n \log n)$, pois:

$$\lim_{n\to\infty} \frac{n \cdot \log_2 n}{n \cdot \log_2 n} = \lim_{n\to\infty} 1 = 1 \in \mathbb{R}_+^*.$$

De forma análoga, podemos dizer que este algoritmo será $\Omega(n \cdot \log_2 n)$ e $\Theta(n \cdot \log_2 n)$.

- **Pior caso**: Como dito anteiormente, nesse caso, a função partição sempre divide a lista de forma desequilibrada, com o pivô sendo o maior ou o menor elemento do vetor. Nesse cenário, uma das sublistas contém apenas um elemento enquanto a outra contém quase todos os demais. Dessa forma, o particionamento acontece com divisões cada vez menores, resultando em uma árvore de recursão linear, como segue:

$$\begin{array}{cc} n & cn \\ n-1 & c(n-1) \\ n-2 & c(n-2) \\ \vdots & \\ 1 & c \end{array}$$

Neste caso, a altura da árvore de recursão é n, já que em cada nível apenas um elemento é fixado na posição correta, e as sublistas vão sendo reduzidas em apenas um elemento a cada iteração.

Assim, o trabalho total realizado por nível é proporcional ao tamanho da lista em cada iteração, somando-se ao longo da altura da árvore. Portanto, a complexidade total é a soma dos primeiros n números, ou seja:

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

Agora, observe que:

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

Assim, podemos afirmar que o tempo de execução do Quick Sort, em sua forma iterativa no pior caso, é $O(n^2)$, $\Omega(n^2)$ e $\Theta(n^2)$

Capítulo 2

Análise de Algoritmo

2.1 Ambiente computacional

Todos os testes de performance nessa seção foram realizados no seguinte sistema:

Software	Sistema Operacional	Arch Linux x86_64
	Kernel	Linux 6.11.5-arch1-1
	Gerenciador de Janelas	Hyprland (Wayland)
	Terminal	Alacritty 0.14.0
	Compilador de C++	clang 18.1.8
	Compilador de Rust	rustc 1.82.0
	Versão do Cargo	cargo 1.82.0
Hardware	CPU	AMD Ryzen 5 5500
	GPU	GeForce RTX 4060 Ti
	Driver da GPU	nvidia (proprietário) 565.57.01
	Memória RAM	31.24 GiB
	Armazenamento	SSD NVMe 2TB

O sistema estava configurado no modo de alto desempenho, com nenhuma outra aplicação rodando além dos testes descritos, e todas as otimizações disponíveis no compilador foram ativadas para garantir uma análise de performance precisa.

2.2 Função iterativa

Ainda na proposta de comparar dois algoritmos que tem a mesma finalidade, o caso em questão diz respeito à duas funções que verificam se o menor elemento de uma lista se repete. Ambas são testadas em listas de tamanho igual a 100, 1000 e 10000, sendo submetidas a cinco testes em cada uma. E a única restrição de entrada é que os elementos dessa lista sejam números entre 1 e 100. Para isso, foram implementados em C++ dois algoritmos que realizam o procedimento descrito. Sendo assim, façamos a análise das diferentes implementações do mesmo algoritmo do ponto de vista do pior, melhor e caso médio.

2.2.1 Algoritmo idadeRep

O algoritmo **idadeRep** foi implementado da seguinte forma

```
1  bool idadeRep(vector<int> &idade) {
2    int tam = idade.size();
3    int indexMenor = 0;
4    for (int i = 1; i < tam; i++) {
5      if (idade[i] < idade[indexMenor]) {
6        indexMenor = i;
7      }
8    }
9    for (int i = 0; i < tam; i++) {
10     if (idade[i] == idade[indexMenor] && i != indexMenor) {
11       return true;
12     }
13   }
14   return false;
15 }
```

2.2.2 Análise do idadeRep

Note que o primeiro loop é inevitável independente do quão favorável seja a caso, e percorrerá a lista por inteira. Apenas no segundo loop que operações podem ser evitadas a depender da lista, pois o retorno de um valor encerra o programa. Além disso, perceba que a implementação acima pode ser visualizada da seguinte forma

$$\begin{aligned} T(n) &= n + n \\ &= 2n \\ &= O(n) \end{aligned}$$

Em suma teremos o seguinte

- **Pior caso:** Sendo o primeiro loop inevitável independente do cenário da lista, o pior caso seria quando o menor elemento só aparece uma vez na lista, portanto seria necessário percorrer a lista por inteira para identificar que ele não se repete.
- **Melhor caso:** Seria quando o menor elemento aparece na primeira e na segunda posição da lista, pois é o caso em que a função retorna na primeira comparação feita.

Sendo assim, temos que $O(n)$, $\Omega(n)$ e $\Theta(n)$.

2.2.3 Resultados do idadeRep

O algoritmo **idadeRep** teve a seguinte performace nos testes

Table 2.1: Tabela de resultados de idadeRep

Lista	n = 100	n = 1000	n = 10000
L1	431 ps	611 ps	3.837 ns
L2	200 ps	521 ps	3.236 ns
L3	251 ps	500 ps	3.186 ns
L4	160 ps	491 ps	3.086 ns
L5	130 ps	531 ps	3.346 ns
Médias	234.4 ps	530.8 ps	3.3382 ns

2.2.4 Algoritmo idadeRep2

Já o algoritmo **idadeRep2** foi implementado da seguinte maneira

```
1 bool idadeRep2(vector<int> &idade) {
2   sort(idade.begin(), idade.end());
3   return idade[0] == idade[1];
4 }
```

2.2.5 Análise do idadeRep2

É possível reparar que a função é relativamente mais enxuta em relação à **idadeRep2**, isso pois sua estratégia para o mesmo fim é diferente. A **idadeRep2** usa uma abordagem que, em um primeiro momento, ordena a lista, e, com ela ordenada, verifica se os dois primeiros elementos são iguais. E de fato, se o objetivo é verificar se o menor elemento da lista se repete, esta se mostra resolutiva para o problema.
Agora note que, em termos de complexidade, o algoritmo se apoia totalmente na função sort da biblioteca std do C++. Também vale a pena falar sobre a implementação da **sort**, que é uma mistura entre Quicksort, Heapsort e Insertion Sort. Portanto, sabemos que a complexidade da **idadeRep2** é igual a da **sort**, que é $O(n * \log_2 n)$ para todos os casos.

2.2.6 Resultados do idadeRep2

Os seguinte resultados foram obtidos

Table 2.2: Tabela de resultados de idadeRep2

Lista	n = 100	n = 1000	n = 10000
L1	2.334 ns	25.969 ns	240.219 ns
L2	2.254 ns	24.425 ns	227.425 ns
L3	2.174 ns	24.317 ns	233.317 ns
L4	2.004 ns	23.414 ns	280.825 ns
L5	2.084 ns	23.243 ns	257.501 ns
Médias	2.17 ns	24.4098 ns	247.857 ns

2.2.7 Comparação dos algoritmos

É possível perceber que o **idadeRep** performa muito melhor em todos os testes. De forma mais específica

- **n = 100:** A média de **idadeRep** é de 234.4 ps, enquanto a de **idadeRep2** é de 2334 ps, ou seja, praticamente 10x mais devagar.
- **n = 10000:** Já nesse, caso **idadeRep** apresenta média de 530.8 ps, enquanto **idadeRep2** apresenta 24409.8 ps, tendo uma diferença de aproximadamente 46x.
- **n = 100000:** Por fim, **idadeRep** tem 3.3382 ns de média. Já **idadeRep2**, 247.857 ns, uma diferença de quase 75x.

Portanto, ao colocar os resultados dos algoritmos lado a lado fica evidente que a implementação de **idadeRep** se sobressai.

2.3 Função Recursiva

Novamente, iremos comparar dois algoritmos que tem a mesmo objetivo, nesse caso, as duas funções buscam um elemento X em uma lista. Ambas são testadas em listas de tamanho igual a 100, 1000 e 10000, sendo submetidas a cinco testes em cada uma. Para isso, foram implementados em C++ dois algoritmos que realizam o procedimento descrito. Sendo assim, façamos a análise das diferentes implementações do mesmo algoritmo do ponto de vista do pior, melhor e caso médio.

2.3.1 Algoritmo buscaBinaria

O algoritmo **buscaBinaria** foi implementado da seguinte forma

```
1  int buscaBinaria(vector<int> &lista, int tamanho, int x) {
2      int esq = 0;
3      int dir = tamanho - 1;
4
5      while (esq < dir) {
6          int m = (esq + dir) / 2;
7          if (x > lista[m]) {
8              esq = m + 1;
9          } else {
10             dir = m;
11         }
12     }
13
14     if (lista[esq] == x) {
15         return esq;
16     }
17
18     return -1;
19 }
```

2.3.2 Análise do buscaBinaria

Note que o loop tem a condição de parada (esq $<$ dir), o que significa que ele só irá terminar quando sobrar somente um valor possível na lista. Portanto, independente do quão favorável seja o caso, ele não irá verificar se lista[m] é o valor que está tentando encontrar, assim executado o loop aproximadamente $\log_2$(tamanho) vezes em todos os casos. Perceba que o algoritmo necessita de melhorias para funcionar de forma que a posição do número buscado seja influente na complexidade do código. Portanto, o loop se comportará da seguinte maneira:

$$n/2^k = 1 \implies n = 2^k \implies k = \log_2 n$$

Em suma teremos o seguinte

- **Casos:** Dado que o loop inevitávelmente rodará $\log_2(tamanho)$ vezes independente do cenário da lista, todos os casos terão a mesma complexidade.

2.3.3 Resultados do buscaBinaria

Portanto, o algoritmo **buscaBinaria** teve a seguinte performace nos testes

Table 2.3: Tabela de resultados de buscaBinaria

Lista	n = 100	n = 1000	n = 10000
L1	190 ps	51 ps	141 ps
L2	80 ps	70 ps	110 ps
L3	80 ps	70 ps	101 ps
L4	50 ps	80 ps	100 ps
L5	50 ps	60 ps	100 ps
Médias	90 ps	66.2 ps	110.4 ps

É perceptivel que a Lista 1 com 100 números foi um outlier em nossa tabela, alterando o valor da média drasticamente tornando-a atípica. Entretanto podemos ver como nos outros casos o tempo é crescente levando em conta o tamanho da lista.

2.3.4 Algoritmo bBinRec

Já o algoritmo **bBinRec** foi implementado da seguinte maneira

```
int bBinRec(vector<int> &lista,int esq, int dir, int x ){
    if (esq>dir) {
        return -1;
    }
    int m = (esq + dir) / 2;
    if (lista[m] == x) {
        return m;
    }
    if (lista[m] > x){
        return bBinRec(lista, esq, m - 1, x);
    }else{
        return bBinRec(lista, m + 1, dir, x);
    }
}
```

2.3.5 Análise do bBinRec

É possível reparar que a função é diferente da **buscaBinaria** pois chama a si mesma em cada passo. Cada chamada reduz o intervalo de busca, mas em vez de usar um loop, ela continua chamando a função com novos valores. Outra diferença é que a função **bBinRec** verifica `lista[m] == x` em cada chamada, permitindo que a função retorne imediatamente se o elemento for encontrado. Portanto, isso mudará a complexidade, tendo casos destintos, diferente do algoritmo anterior. Em suma teremos o seguinte:

- **Melhor Caso:** $O(1)$ — o elemento está no meio na primeira chamada.
- **Caso Médio:** $O(\log_2 n)$ — o elemento está em uma posição aleatória e requer, em média, $\log_2 n$ divisões.
- **Pior Caso:** $O(\log_2 n)$ — o elemento não está presente ou está no final da lista, e o algoritmo precisa dividir o intervalo até se esgotar.

2.3.6 Resultados do bBinRec

A versão recursiva é ligeiramente mais rápida na prática, pois pode interromper o processo assim que encontra o elemento, sem precisar fazer verificações adicionais no intervalo. Portanto, o algoritmo **bBinRec** teve a seguinte performace nos testes:

Table 2.4: Tabela de resultados de bBinRec

Lista	n = 100	n = 1000	n = 10000
L1	80 ps	60 ps	70 ps
L2	40 ps	51 ps	71 ps
L3	60 ps	50 ps	50 ps
L4	50 ps	50 ps	60 ps
L5	50 ps	50 ps	60 ps
Médias	56 ps	52.2 ps	62.2 ps

Novamente, perceptivel que a Lista 1 com 100 números foi um outlier em nossa tabela, alterando o valor da média drasticamente tornando-a atípica. Entretanto podemos ver como nos outros casos o tempo é crescente levando em conta o tamanho da lista.

2.3.7 Comparação dos algoritmos

É possível perceber que o **bBinRec** performa melhor em todos os testes. De forma mais específica

- **n = 100:** A média de **buscaBinaria** é de 90 ps, enquanto a de **bBinRec** é de 56 ps.
- **n = 10000:** Já nesse, caso **buscaBinaria** apresenta média de 66,2 ps, enquanto **bBinRec** apresenta 52,2 ps.
- **n = 100000:** Por fim, **buscaBinaria** tem 110.4 ps de média. Já **bBinRec**, 62,2 ps.

Assim, ao comparar os resultados dos algoritmos lado a lado, torna-se evidente que a implementação de **bBinRec** se destaca.

2.4 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.

2.4.1 Bubble Sort

Iterativo

```rust
pub fn iterative_bubble_sort<T: PartialOrd>(arr: &mut [T]) {
  for i in 0..arr.len() {
    for j in 1..arr.len() - i {
      if arr[j - 1] > arr[j] {
        arr.swap(j - 1, j);
      }
    }
  }
}
```

Recursivo

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

```rust
pub fn recursive_bubble_sort<T: PartialOrd>(arr: &mut [T]) {
  if arr.is_empty() {
    return;
  }
  let last_element_position = arr.len();
  bubble_sort_pass(arr, 1, last_element_position);
  recursive_bubble_sort(&mut arr[..last_element_position - 1]);
}
```

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 inicio ao fim.

```rust
fn bubble_sort_pass<T: PartialOrd>(arr: &mut [T], iterator: usize,
    last_element_position: usize) {
  if iterator >= last_element_position {
    return;
  }
  if arr[iterator - 1] > arr[iterator] {
    arr.swap(iterator - 1, iterator)
  }
  bubble_sort_pass(arr, iterator + 1, last_element_position)
}
```

2.4.2 Quick Sort

Iterativo

Como o Quick Sort é um algoritmo naturalmente recursivo, precisávamos 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.

```rust
pub fn iterative_quick_sort<T: PartialOrd + Copy>(arr: &mut [T]) {
  if arr.is_empty() {
    return;
  }
  let mut stack = vec![(0, arr.len() - 1)];
  while let Some((low, high)) = stack.pop() {
```

```
    if low < high {
      let pivot_index = partition(arr, low, high);
      if pivot_index > 0 {
        stack.push((low, pivot_index - 1)); // Left side
      }
      stack.push((pivot_index + 1, high)); // Right side
    }
  }
}
```

Recursivo

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

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

```rust
fn _recursive_quick_sort<T: PartialOrd + Copy>(
  arr: &mut [T],
  lower_bound: usize,
  upper_bound: usize,
) {
  if lower_bound >= upper_bound {
    return;
  }
  let pivot_index = partition(arr, lower_bound, upper_bound);
  if pivot_index > 0 {
    _recursive_quick_sort(arr, lower_bound, pivot_index - 1);
  }
  _recursive_quick_sort(arr, pivot_index + 1, upper_bound);
}
```

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 pivó 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.

```rust
fn partition<T: PartialOrd + Copy>(arr: &mut [T], lower_bound: usize,
    upper_bound: usize) -> usize {
  let pivot = arr[upper_bound];
  let mut left_item = lower_bound as isize - 1;
  for right_item in lower_bound..upper_bound {
    if arr[right_item] < pivot {
      left_item += 1;
      arr.swap(left_item as usize, right_item);
    }
  }
  let new_pivot = (left_item + 1) as usize;
  arr.swap(new_pivot, upper_bound);
  new_pivot
}
```

2.4.3 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.

```rust
pub fn iterative_merge_sort<T: PartialOrd + Copy>(arr: &mut [T]) {
  if arr.len() <= 1 {
    return;
  }
  let mut temp_arr = arr.to_vec();
  let mut segment_size = 1;
  let arr_len = arr.len();
  while segment_size < arr.len() {
    let mut start = 0;
```

```
    while start < arr_len {
      let mid = (start + segment_size).min(arr_len);
      let end = (start + 2 * segment_size).min(arr_len);
      merge(&mut temp_arr[start..end], &arr[start..mid], &arr[mid..end]);
      start += 2 * segment_size;
    }
    arr.copy_from_slice(&temp_arr);
    segment_size *= 2;
  }
}
```

Recursivo

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

```
pub fn recursive_merge_sort<T: PartialOrd + Copy>(arr: &mut [T]) {
  if arr.len() <= 1 {
    return;
  }
  let mid = arr.len() / 2;
  let mut left_arr = arr[..mid].to_vec();
  let mut right_arr = arr[mid..].to_vec();
  recursive_merge_sort(&mut left_arr);
  recursive_merge_sort(&mut right_arr);
  merge(arr, &left_arr, &right_arr);
}
```

Função auxiliar

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

```
fn merge<T: PartialOrd + Copy>(arr: &mut [T], left_arr: &[T], right_arr:
    &[T]) {
  let left_arr_len = left_arr.len();
  let right_arr_len = right_arr.len();
  let (mut i, mut l, mut r) = (0, 0, 0);
  while l < left_arr_len && r < right_arr_len {
    if left_arr[l] < right_arr[r] {
      arr[i] = left_arr[l];
      l += 1;
    } else {
      arr[i] = right_arr[r];
      r += 1;
    }
    i += 1;
  }
  while l < left_arr_len {
    arr[i] = left_arr[l];
    i += 1;
    l += 1;
  }
  while r < right_arr_len {
    arr[i] = right_arr[r];
    i += 1;
    r += 1;
  }
}
```

2.4.4 Organização do projeto e corretude do algoritmo

Para testar os algoritmos o projeto foi subdivido 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 depêndencia usada foi rand, para poder gerar números aleatórios de 0 a 100000

```
[dependencies]
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:

```
fn reverse_list_test(func: fn(&mut [i32])) {
  let mut arr = [5, 3, 2, 4, 1];
  func(&mut arr);
  assert_eq!(arr, [1, 2, 3, 4, 5], "reverse_list_test failed");
}

fn duplicates_list_test(func: fn(&mut [i32])) {
  let mut arr = [4, 2, 3, 2, 1, 4];
  func(&mut arr);
  assert_eq!(arr, [1, 2, 2, 3, 4, 4], "duplicates_list_test failed");
}

fn already_sorted_list_test(func: fn(&mut [i32])) {
  let mut arr = [1, 2, 3, 4, 5];
  func(&mut arr);
  assert_eq!(arr, [1, 2, 3, 4, 5], "already_sorted_list_test failed");
}

fn singleton_list_test(func: fn(&mut [i32])) {
  let mut arr = [42];
  func(&mut arr);
  assert_eq!(arr, [42], "singleton_list_test faild");
}

fn empty_list_test(func: fn(&mut [i32])) {
  let mut arr: [i32; 0] = [];
  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`

2.5 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 2.5: 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 instanciação 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 (2024*a*), '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 (2024*b*), 'Heapsort — wikipedia, the free encyclopedia'. [Acessado: 05 de Novembro, 2024].
URL: *https://en.wikipedia.org/wiki/Heapsort*

Contribuidores da Wikipedia (2024*c*), 'Insertion sort — wikipedia, the free encyclopedia'. [Acessado: 05 de Novembro, 2024].
URL: *https://en.wikipedia.org/wiki/Insertion_sort*

Contribuidores da Wikipedia (2024*d*), 'John von neumann — wikipedia, the free encyclopedia'. [Acessado: 26 de Outubro, 2024].
URL: *https://pt.wikipedia.org/wiki/John_von_Neumann*

Contribuidores da Wikipedia (2024*e*), 'Merge sort'. [Acessado: 26 de Outubro, 2024].
URL: *https://en.wikipedia.org/wiki/Merge_sort*

Contribuidores da Wikipedia (2024*f*), 'Quicksort — wikipedia, the free encyclopedia'. [Acessado: 26 de Outubro, 2024].
URL: *https://en.wikipedia.org/wiki/Quicksort*

Contribuidores do CppReference (2024), 'std::sort'. [Acessado: 05 de Novembro, 2024].
URL: *https://en.cppreference.com/w/cpp/algorithm/sort*

Contribuidores do GeeksforGeeks (2024*a*), 'Merge sort'. [Acessado: 26 de Outubro, 2024].
URL: *https://www.geeksforgeeks.org/merge-sort/*

Contribuidores do GeeksforGeeks (2024*b*), '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/*