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

27 de outubro de 2024

Sumário

Capítulo 1

Análise Teórica

1.1 Bubble Sort

TODO

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.

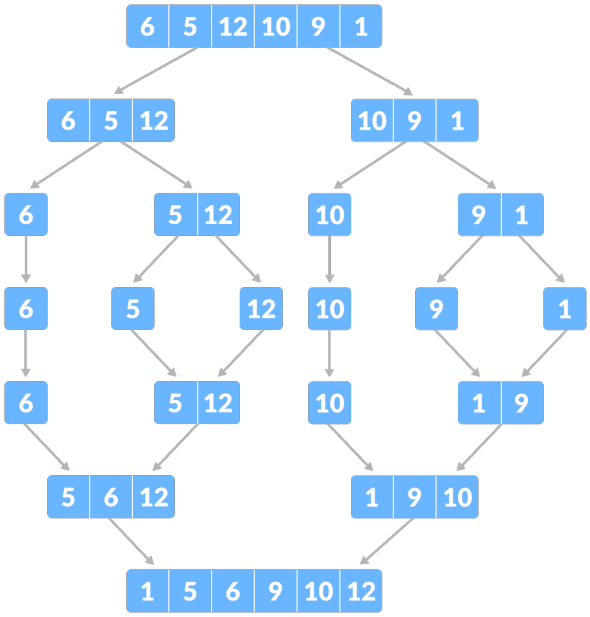


Figure 1.1: Diagrama exemplo de um Merge Sort

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: **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(1^a metade da **lista**), MergeSort(2^a 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:

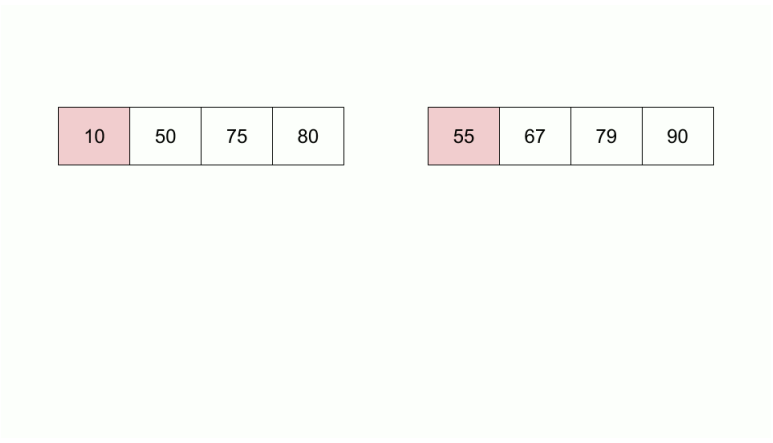


Figure 1.2: Comparando o primeiro elemento da primeira lista com o primeiro da segunda

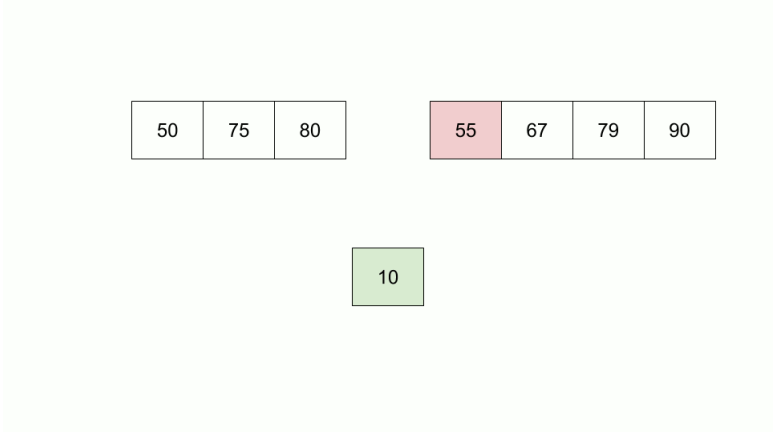


Figure 1.3: 10 é menor que 55, então é anexado no final da lista resultado

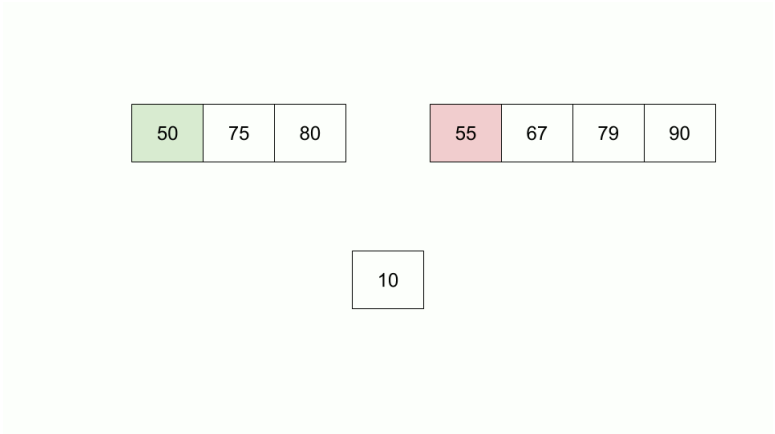


Figure 1.4: Comparando o próximo elemento da primeira lista

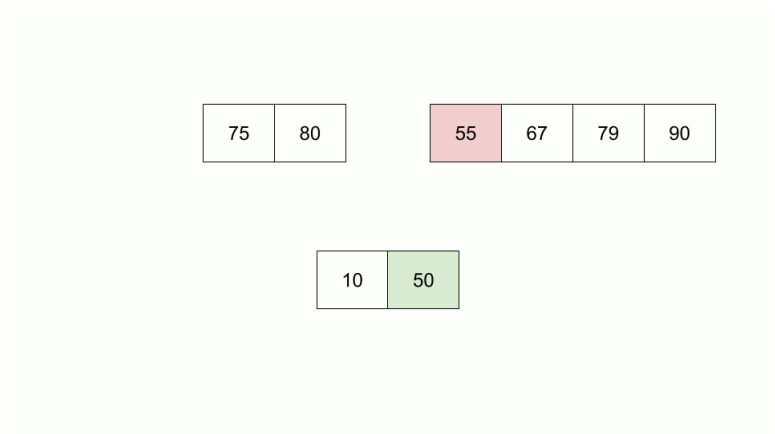


Figure 1.5: 50 é menor que 55, então é anexado no final da lista resultado

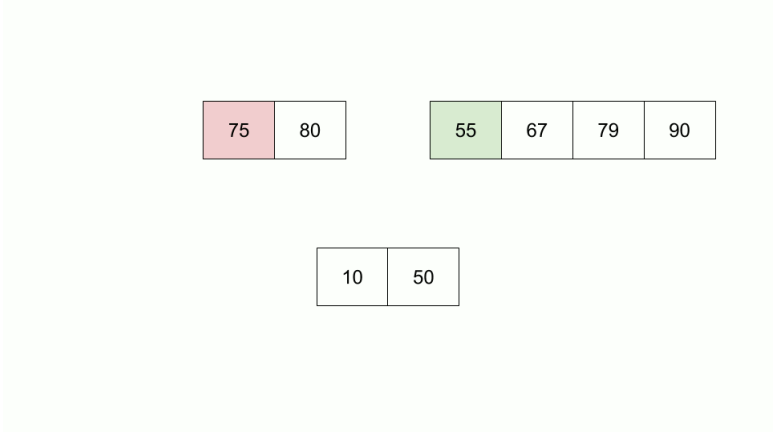


Figure 1.6: Comparando o próximo elemento da primeira lista

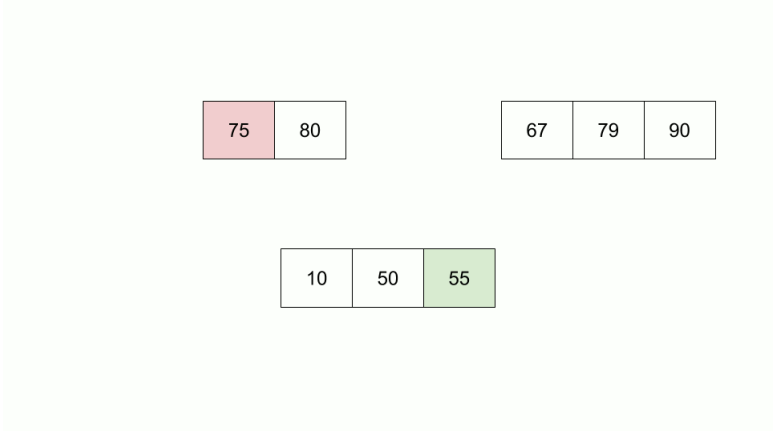


Figure 1.7: 55 é menor, então é anexado no final da lista resultado

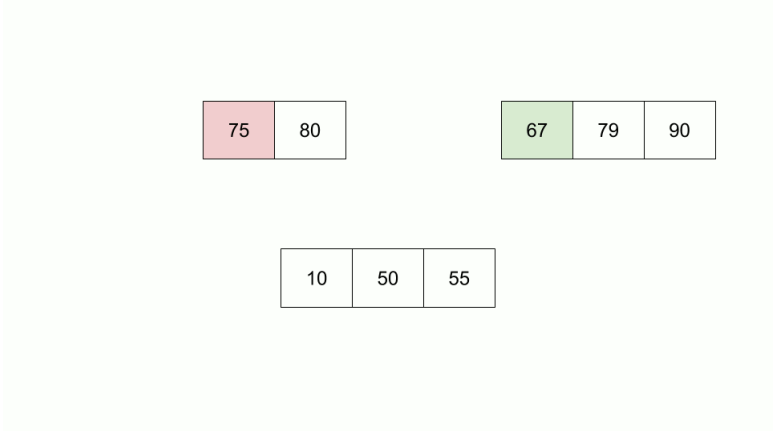


Figure 1.8: Comparando o próximo elemento da segunda lista

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 $\Omega(n)$.

Análise da versão iterativa

TODO

Análise da versão recursiva

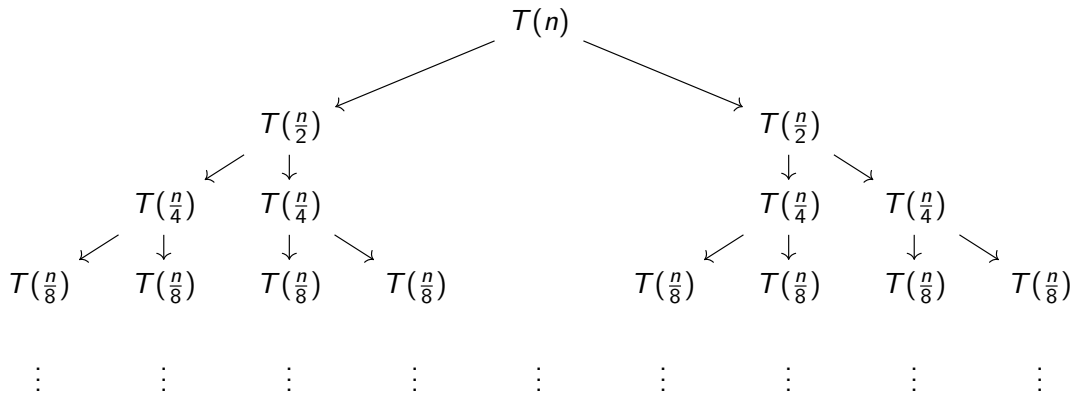
Analisando o corpo da função, teremos dois casos: caso o tamanho da lista (n) seja menor ou igual 1 e caso contrário.

```
if tamanho da lista ≤ 1 then return
end if
return Merge(MergeSort(1ª metade da lista), MergeSort(2ª metade da lista)
```

No primeiro caso, é facil 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, vamos usar primeiro a **Árvore de recorrência** como método. Comecemos 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:

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

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 $\Omega(n \cdot \log_2 n)$.

1.3 Quick Sort

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

```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)
}
```

2.3.2 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.

```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.

```rust
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
}
```

2.3.3 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.

```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 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;
  }
}
```

2.3.4 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`

2.4 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 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*), '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*c*), 'Merge sort'. [Acessado: 26 de Outubro, 2024].
URL: *https://en.wikipedia.org/wiki/Merge_sort*

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

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/*