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

26 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.

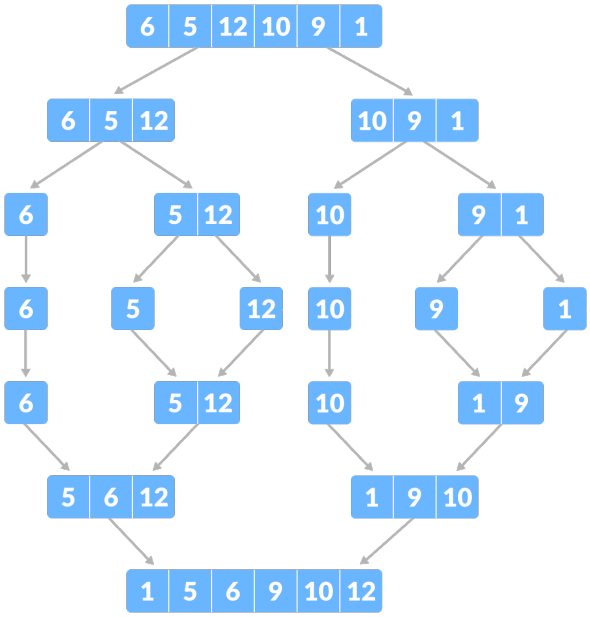

Figure 1.1: Diagrama exemplo de um Merge Sort

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:

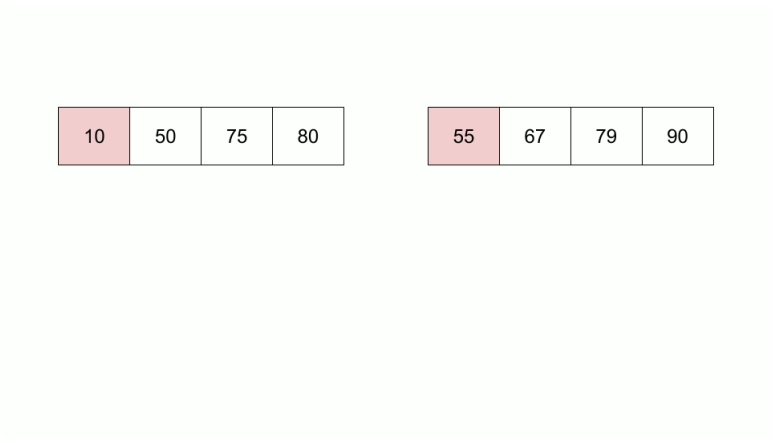

Figure 1.2: Comparando o primeiro elemento da primeira lista com o primeiro da segunda

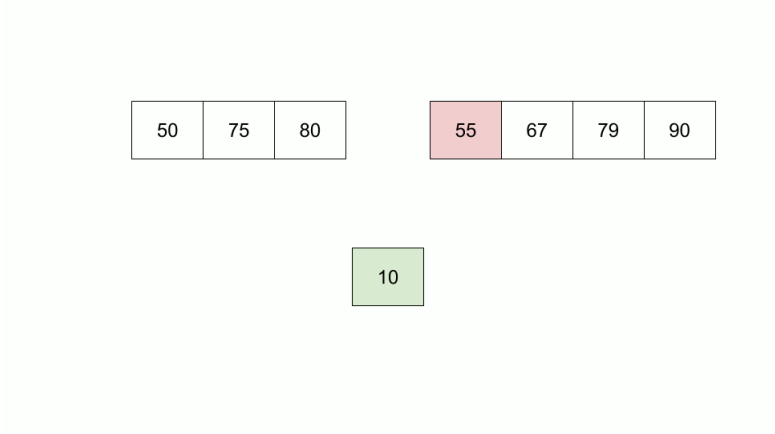

Figure 1.3: 10 é menor que 55, então é anexado no final da lista resultado

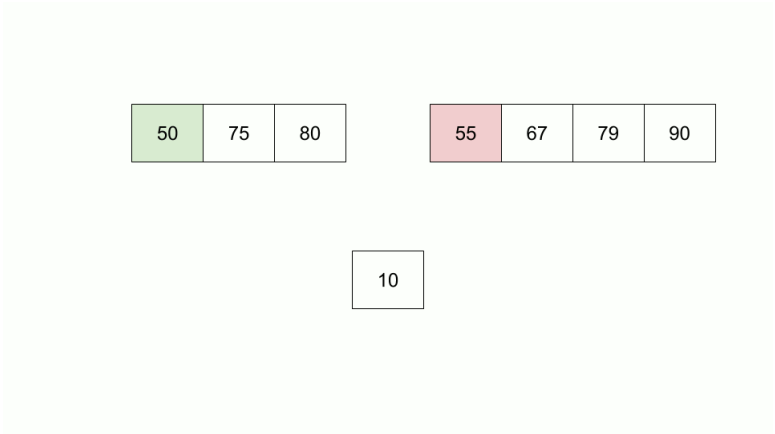

Figure 1.4: Comparando o próximo elemento da primeira lista

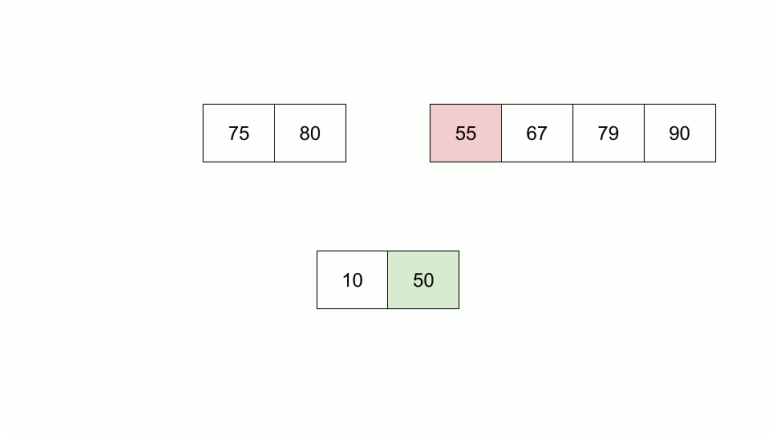

Figure 1.5: 50 é menor que 55, então é anexado no final da lista resultado

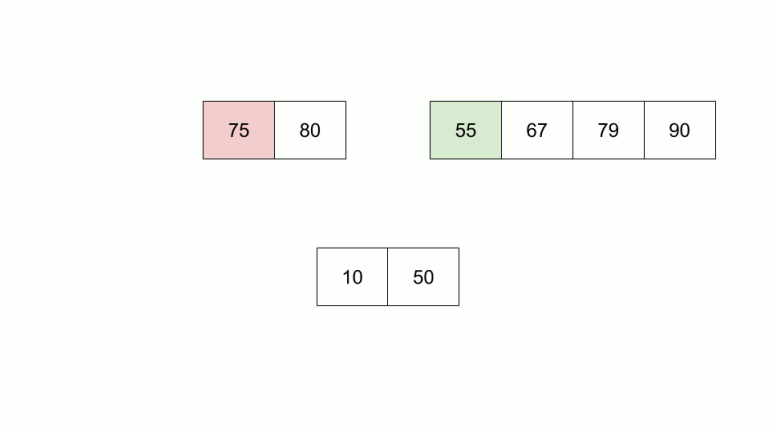

Figure 1.6: Comparando o próximo elemento da primeira lista

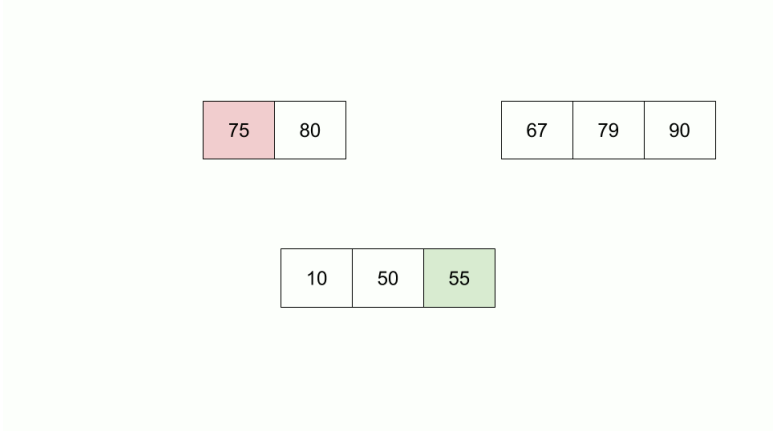

Figure 1.7: 55 é menor, então é anexado no final da lista resultado

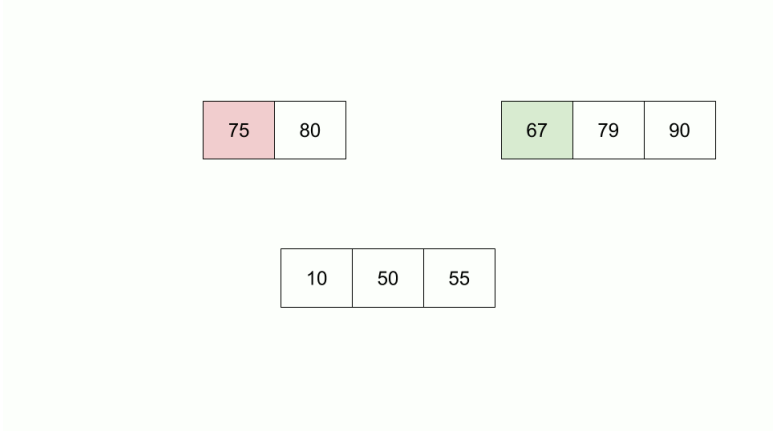

Figure 1.8: Comparando o próximo elemento da segunda lista

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

Dada as duas versões do *Merge sort*, analisemos então suas respectivas complexidades.

Análise da versão iterativa

TODO

Análise da versão recursiva

TODO

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.

Capítulo 3

Methodology

We mentioned in Chapter 1 that a project report's structure could follow a particular paradigm. Hence, the organization of a report (effectively the Table of Content of a report) can vary depending on the type of project you are doing. Check which of the given examples suit your project. Alternatively, follow your supervisor's advice.

3.1 Examples of the sections of a methodology chapter

A general report structure is summarised (suggested) in Table 3.1. Table 3.1 describes that, in general, a typical report structure has three main parts: (1) front matter, (2) main text, and (3) end matter. The structure of the front matter and end matter will remain the same for all the undergraduate final year project report. However, the main text varies as per the project's needs.

Table 3.1: Undergraduate report template structure

Frontmatter		Title Page Abstract Acknowledgements Table of Contents List of Figures List of Tables List of Abbreviations
Main text	Chapter 1	Introduction
	Chapter 2	Literature Review
	Chapter 3	Methodology
	Chapter 4	Results
	Chapter 5	Discussion and Analysis
	Chapter 6	Conclusions and Future Work
	Chapter 7	Refection
End matter		References Appendices (Optional) Index (Optional)

3.1.1 Example of a software/Web development main text structure

Notice that the "methodology" Chapter of Software/Web development in Table 3.2 takes a standard software engineering paradigm (approach). Alternatively, these suggested sections can be the chapters of their own. Also, notice that "Chapter 5" in Table 3.2 is "Testing and Validation" which is different from the general report template mentioned in Table 3.1. Check with your supervisor if in doubt.

Table 3.2: Example of a software engineering-type report structure

Chapter 1	Introduction	
Chapter 2	Literature Review	
Chapter 3	Methodology	
		Requirements specifications
		Analysis
		Design
		Implementations
Chapter 4	Testing and Validation	
Chapter 5	Results and Discussion	
Chapter 6	Conclusions and Future Work	
Chapter 7	Reflection	

3.1.2 Example of an algorithm analysis main text structure

Some project might involve the implementation of a state-of-the-art algorithm and its performance analysis and comparison with other algorithms. In that case, the suggestion in Table 3.3 may suit you the best.

Table 3.3: Example of an algorithm analysis type report structure

Chapter 1	Introduction	
Chapter 2	Literature Review	
Chapter 3	Methodology	
		Algorithms descriptions
		Implementations
		Experiments design
Chapter 4	Results	
Chapter 5	Discussion and Analysis	
Chapter 6	Conclusion and Future Work	
Chapter 7	Reflection	

3.1.3 Example of an application type main text structure

If you are applying some algorithms/tools/technologies on some problems/datasets/etc., you may use the methodology section prescribed in Table 3.4.

Table 3.4: Example of an application type report structure

Chapter 1	Introduction	
Chapter 2	Literature Review	
Chapter 3	Methodology	
		Problems (tasks) descriptions
		Algorithms/tools/technologies/etc. descriptions
		Implementations
		Experiments design and setup
Chapter 4	Results	
Chapter 5	Discussion and Analysis	
Chapter 6	Conclusion and Future Work	
Chapter 7	Reflection	

3.1.4 Example of a science lab-type main text structure

If you are doing a science lab experiment type of project, you may use the methodology section suggested in Table 3.5. In this kind of project, you may refer to the "Methodology" section as "Materials and Methods."

Table 3.5: Example of a science lab experiment-type report structure

Chapter 1	Introduction	
Chapter 2	Literature Review	
Chapter 3	Materials and Methods	
		Problems (tasks) description
		Materials
		Procedures
		Implementations
		Experiment set-up
Chapter 4	Results	
Chapter 5	Discussion and Analysis	
Chapter 6	Conclusion and Future Work	
Chapter 7	Reflection	

3.2 Example of an Equation in LaTeX

Eq. 3.1 [note that this is an example of an equation's in-text citation] is an example of an equation in LaTeX. In Eq. (3.1), s is the mean of elements $x_i \in \mathbf{x}$:

$$s = \frac{1}{N} \sum_{i=1}^{N} x_i. \tag{3.1}$$

Have you noticed that all the variables of the equation are defined using the **in-text** maths command $.$, and Eq. (3.1) is treated as a part of the sentence with proper punctuation? Always treat an equation or expression as a part of the sentence.

3.3 Example of a Figure in LaTeX

Figure 3.1 is an example of a figure in LaTeX. For more details, check the link:
wikibooks.org/wiki/LaTeX/Floats,_Figures_and_Captions.
Keep your artwork (graphics, figures, illustrations) clean and readable. At least 300dpi is a good resolution of a PNG format artwork. However, an SVG format artwork saved as a PDF will produce the best quality graphics. There are numerous tools out there that can produce vector graphics and let you save that as an SVG file and/or as a PDF file. One example of such a tool is the "Flow algorithm software". Here is the link for that: flowgorithm.org.

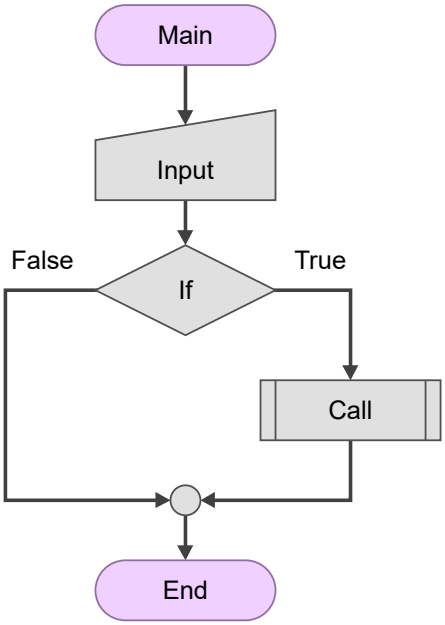

Figure 3.1: Example figure in LaTeX.

3.4 Example of an algorithm in LaTeX

Algorithm 1 is a good example of an algorithm in LaTeX.

Algorithm 1 Example caption: sum of all even numbers

Input: $\mathbf{x} = x_1, x_2, \ldots, x_N$
Output: $EvenSum$ (Sum of even numbers in $\mathbf{x}$)

```
1: function EvenSummation(x)
2:     EvenSum ← 0
3:     N ← length(x)
4:     for i ← 1 to N do
5:         if x_i mod 2 == 0 then          ▷ Check whether a number is even.
6:             EvenSum ← EvenSum + x_i
7:         end if
8:     end for
9:     return EvenSum
10: end function
```

3.5 Example of code snippet in LaTeX

Code Listing 3.1 is a good example of including a code snippet in a report. While using code snippets, take care of the following:

- do not paste your entire code (implementation) or everything you have coded. Add code snippets only.
- The algorithm shown in Algorithm 1 is usually preferred over code snippets in a technical/scientific report.
- Make sure the entire code snippet or algorithm stays on a single page and does not overflow to another page(s).

Here are three examples of code snippets for three different languages (Python, Java, and CPP) illustrated in Listings 3.1, 3.2, and 3.3 respectively.

```python
import numpy as np

x  = [0, 1, 2, 3, 4, 5] # assign values to an array
evenSum = evenSummation(x) # call a function

def evenSummation(x):
    evenSum = 0
    n = len(x)
    for i in range(n):
        if np.mod(x[i],2) == 0: # check if a number is even?
            evenSum = evenSum + x[i]
    return evenSum
```

Listing 3.1: Code snippet in LaTeX and this is a Python code example

Here we used the "\clearpage" command and forced-out the second listing example onto the next page.

```java
public class EvenSum{
    public static int evenSummation(int[] x){
        int evenSum = 0;
        int n = x.length;
        for(int i = 0; i < n; i++){
            if(x[i]%2 == 0){ // check if a number is even?
                evenSum = evenSum + x[i];
            }
        }
        return evenSum;
    }
    public static void main(String[] args){
        int[] x  = {0, 1, 2, 3, 4, 5}; // assign values to an array
        int evenSum = evenSummation(x);
        System.out.println(evenSum);
    }
}
```

Listing 3.2: Code snippet in LaTeX and this is a Java code example

```cpp
int evenSummation(int x[]){
    int evenSum = 0;
    int n = sizeof(x);
    for(int i = 0; i < n; i++){
        if(x[i]%2 == 0){ // check if a number is even?
            evenSum = evenSum + x[i];
      }
    }
    return evenSum;
}

int main(){
    int x[]  = {0, 1, 2, 3, 4, 5}; // assign values to an array
    int evenSum = evenSummation(x);
    cout<<evenSum;
    return 0;
}
```

Listing 3.3: Code snippet in LaTeX and this is a C/C++ code example

3.6 Example of in-text citation style

3.6.1 Example of the equations and illustrations placement and reference in the text

Make sure whenever you refer to the equations, tables, figures, algorithms, and listings for the first time, they also appear (placed) somewhere on the same page or in the following page(s). Always make sure to refer to the equations, tables and figures used in the report. Do not leave them without an **in-text citation**. You can refer to equations, tables and figures more them once.

3.6.2 Example of the equations and illustrations style

Write **Eq.** with an uppercase "Eq" for an equation before using an equation number with (\eqref{.}). Use "Table" to refer to a table, "Figure" to refer to a figure, "Algorithm" to refer to an algorithm and "Listing" to refer to listings (code snippets). Note that, we do not use

the articles "a," "an," and "the" before the words Eq., Figure, Table, and Listing, but you may use an article for referring the words figure, table, etc. in general.

For example, the sentence "A report structure is shown in **the** Table 3.1" should be written as "A report structure is shown **in** Table 3.1."

3.7 Summary

Write a summary of this chapter.

Note: In the case of **software engineering** project a Chapter "**Testing and Validation**" should precede the "Results" chapter. See Section 3.1.1 for report organization of such project.

Capítulo 4

Results

The results chapter tells a reader about your findings based on the methodology you have used to solve the investigated problem. For example:

- If your project aims to develop a software/web application, the results may be the developed software/system/performance of the system, etc., obtained using a relevant methodological approach in software engineering.
- If your project aims to implement an algorithm for its analysis, the results may be the performance of the algorithm obtained using a relevant experiment design.
- If your project aims to solve some problems/research questions over a collected dataset, the results may be the findings obtained using the applied tools/algorithms/etc.

Arrange your results and findings in a logical sequence.

4.1 A section

...

4.2 Example of a Table in LaTeX

Table 4.1 is an example of a table created using the package LaTeX"booktabs." do check the link: wikibooks.org/wiki/LaTeX/Tables for more details. A table should be clean and readable. Unnecessary horizontal lines and vertical lines in tables make them unreadable and messy. The example in Table 4.1 uses a minimum number of liens (only necessary ones). Make sure that the top rule and bottom rule (top and bottom horizontal lines) of a table are present.

Table 4.1: Example of a table in LaTeX

Bike		
Type	Color	Price (£)
Electric	black	700
Hybrid	blue	500
Road	blue	300
Mountain	red	300
Folding	black	500

4.3 Example of captions style

- The **caption of a Figure (artwork) goes below** the artwork (Figure/Graphics/illustration). See example artwork in Figure 3.1.
- The **caption of a Table goes above** the table. See the example in Table 4.1.
- The **caption of an Algorithm goes above** the algorithm. See the example in Algorithm 1.
- The **caption of a Listing goes below** the Listing (Code snippet). See example listing in Listing 3.1.

4.4 Summary

Write a summary of this chapter.

Capítulo 5

Discussion and Analysis

Depending on the type of project you are doing, this chapter can be merged with "Results" Chapter as " Results and Discussion" as suggested by your supervisor.

In the case of software development and the standalone applications, describe the significance of the obtained results/performance of the system.

5.1 A section

The Discussion and Analysis chapter evaluates and analyses the results. It interprets the obtained results.

5.2 Significance of the findings

In this chapter, you should also try to discuss the significance of the results and key findings, in order to enhance the reader's understanding of the investigated problem

5.3 Limitations

Discuss the key limitations and potential implications or improvements of the findings.

5.4 Summary

Write a summary of this chapter.

Capítulo 6

Conclusions and Future Work

6.1 Conclusions

Typically a conclusions chapter first summarizes the investigated problem and its aims and objectives. It summaries the critical/significant/major findings/results about the aims and objectives that have been obtained by applying the key methods/implementations/experiment set-ups. A conclusions chapter draws a picture/outline of your project's central and the most signification contributions and achievements.

A good conclusions summary could be approximately 300–500 words long, but this is just a recommendation.

A conclusions chapter followed by an abstract is the last things you write in your project report.

6.2 Future work

This section should refer to Chapter 4 where the author has reflected their criticality about their own solution. The future work is then sensibly proposed in this section.

Guidance on writing future work: While working on a project, you gain experience and learn the potential of your project and its future works. Discuss the future work of the project in technical terms. This has to be based on what has not been yet achieved in comparison to what you had initially planned and what you have learned from the project. Describe to a reader what future work(s) can be started from the things you have completed. This includes identifying what has not been achieved and what could be achieved.

A good future work summary could be approximately 300–500 words long, but this is just a recommendation.

Capítulo 7

Reflection

Write a short paragraph on the substantial learning experience. This can include your decision-making approach in problem-solving.

Some hints: You obviously learned how to use different programming languages, write reports in LaTeXand use other technical tools. In this section, we are more interested in what you thought about the experience. Take some time to think and reflect on your individual project as an experience, rather than just a list of technical skills and knowledge. You may describe things you have learned from the research approach and strategy, the process of identifying and solving a problem, the process research inquiry, and the understanding of the impact of the project on your learning experience and future work.

Also think in terms of:

- what knowledge and skills you have developed
- what challenges you faced, but was not able to overcome
- what you could do this project differently if the same or similar problem would come
- rationalize the divisions from your initial planed aims and objectives.

A good reflective summary could be approximately 300–500 words long, but this is just a recommendation.

Note: The next chapter is "**References**," which will be automatically generated if you are using BibTeX referencing method. This template uses BibTeX referencing. Also, note that there is difference between "References" and "Bibliography." The list of "References" strictly only contain the list of articles, paper, and content you have cited (i.e., refereed) in the report. Whereas Bibliography is a list that contains the list of articles, paper, and content you have cited in the report plus the list of articles, paper, and content you have read in order to gain knowledge from. We recommend to use only the list of "References."

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/*