

Universidade de Brasília  
Instituto de Ciências Exatas  
Departamento de Ciência da Computação

117536 - Projeto e Análise de Algoritmos  
Turma: B

Análise Assintótica e Corretude do Algoritmo  
MergeSort Utilizando PVS

Gabriel Levi - 16/0006490  
Gabriel Nunes - 16/0006597

4 de dezembro de 2019

# 1 Introdução

A verificação formal de algoritmo, no que tange ao seu comportamento assintótico e a corretude do algoritmo é interesse central da Ciência da Computação. Por vezes, a prova via argumentação, isto é, lápis e papel pode ser suficiente para que o escritor convença o leitor de que o algoritmo está correto. Contudo, esse modelo de prova se sustenta muita vezes em passos de pura intuição, assumições que ambas as partes enxergam como um axioma, e saltos lógicos que por mais naturais que pareçam escondem um conjunto não-trivial de conceitos. A ocorrência desses aspectos em uma formalização pode ocultar falhas que, de fato, provem a incorretude do algoritmo e desmonstre um comportamento assintótico pior do que o esperado.

Como forma de minimizar o exposto anteriormente, introduz-se os sistemas de verificação de provas. Os verificadores garantem que os passos realizados dentro de uma prova respeitem o conjunto de regras de sua lógica intrínseca. Então, dadas premissas corretas e um ponto factível onde se deseja chegar, qualquer passo intermediário tem que, necessariamente, estar correto. Obviamente, construções ruins de objetivos e premissas podem levar a provas, ainda sim, incorretas ou impossíveis. Podemos concluir então que os sistemas de verificação pressupõe que uma prova, ou pelo menos a ideia da mesma, já exista e o usuário interessado o utilize para demonstrar que de fato aquela construção vale.

Um dos verificadores, PVS - Prototype Verification System - é a linguagem de especificação e provador automatizado de teoremas que aqui será utilizado. PVS trabalha com implementações em distribuições de diferentes versões de LISP. Em um arquivo, o usuário define premissas - um algoritmo - e teoremas. O arquivo é dado como entrada para o provador que requisita as regras a serem aplicadas até que o teorema desejado seja provado. As regras reconhecidas pelo PVS tratam-se regras do cálculo de seqüentes. O PVS permite também que uma prova possa ser revisitada em uma representação gráfica que revela cada passo bem como suas dependências.

O objetivo deste trabalho é analisar assintoticamente o custo de tempo e a corretude do algoritmo de ordenação *Merge Sort* via PVS. O Merge Sort foi criado em meados de 1945 por John Von Neumann. A escolha deste algoritmo se deu por ser de implementação muito simples para múltiplas estruturas de dados tais como vetores e listas ligadas e por, ainda sim, ser um algoritmo com múltiplas aplicações dado o seu custo de execução. Este relatório apresentará a ideia de prova formalizada via provador e fica a critério do leitor visitar

o repositório para verificar a mesma. A formalização completa pode ser encontrada em [github.com/paa-2019-2/levi-nunes](https://github.com/paa-2019-2/levi-nunes).

Esse documento se organizará, daqui em diante, por um capítulo 2 de revisão teórica. Em seguida, no capítulo 3, a apresentação do algoritmo Merge Sort. O capítulo 5 apresentará a ideia abordada pelos autores para a análise de assintótica do algoritmo bem como argumentação sobre o pior e o melhor caso do mesmo. O capítulo 4 apresentará, como o capítulo anterior, a ideia abordada e sua argumentação. Por fim, o capítulo de conclusão, apresentará um resumo dos resultados aqui encontrados acompanhado de um meta-texto discorrendo sobre as principais dificuldades do trabalho com o verificador formal de provas.

## 2 Revisão teórica

Este capítulo introduzirá conceitos necessários e notações necessárias para o entedimento das seções posteriores.

### 2.1 Notação assintótica

A notação assintótica é uma forma de descrever o comportamento de uma função matemática para um  $n$  inteiros suficientemente grandes. Neste trabalho, interessa-se somente funções eventualmente crescentes uma vez que a intenção é analisar custo de execução de algoritmos. Segue-se então as definições:

**Definição 2.1.**  $O(g(n)) := \{f(n) \mid 0 \leq f(n) \leq c * g(n), c \text{ constante e } n \geq n_0\}$

Uma função  $f(n)$  que descreve o custo de um algoritmo  $A$  pertence à  $O(g(n))$  se, e somente se, para entradas suficientemente grandes o custo do pior caso do algoritmo é limitado superiormente por uma função da ordem de  $g(n)$ . Isto quer dizer independente da entrada  $n$ ,  $n \geq n_0$  o algoritmo nunca tomará mais que  $c * g(n)$  passos,  $c$  constante, para concluir sua execução.

**Definição 2.2.**  $\Omega(g(n)) := \{f(n) \mid 0 \leq c * g(n) \leq f(n), c \text{ constante e } n \geq n_0\}$

A definição de  $\Omega$  é similar a definição de  $O$ , com a diferença que  $\Omega$  descreve o comportamento do algoritmo em seu melhor caso. Isto é, independente da entrada  $n$ ,  $n \geq n_0$  o algoritmo tomará ao menos  $c * g(n)$  passos,  $c$  constante, para concluir sua execução.

**Definição 2.3.**  $\Theta(g(n)) := \{f(n) \mid f(n) \in \Omega(g(n)) \wedge f(n) \in O(g(n))\}$

Por último, a definição de  $\Theta$  descreve uma classe de algoritmos em que o melhor e o pior caso estão na mesma ordem de complexidade. É uma notação mais precisa em relação a complexidade do algoritmo mas que, efetivamente, não se aplica a qualquer análise.

## 2.2 Lista

Uma lista é uma estrutura recursiva que obedece a seguinte gramática:

$$list := nil \mid a :: list$$

Onde *nil* representa uma lista vazia,  $a :: list$  representa um elemento qualquer concatenado com uma lista. À fim de simplicidade, o primeiro elemento de uma lista  $l$  é denotado aqui como  $car(l)$  e a lista restante, a calda, é denotada como  $cdr(l)$ . Também para fim de simplicidade, denota-se para uma lista  $l$ , qualquer,  $LENGTH(l)$  a quantidade de elementos que a mesma armazena.

## 3 O algoritmo Merge Sort

O algoritmo fundamenta-se na técnica Dividir-para-Conquistar. A técnica consiste em, dada uma instância do problema quebra-la em partes até que as mesmas sejam fáceis ou triviais de se resolver, por fim, cada pequena solução é combinada para de obter a solução para o problema maior. Apesar de parecer similar, esta técnica é bem diferente de programação dinâmica pois cada subproblema é disjunto dos demais. Desta forma, o algoritmo recebe uma lista como entrada uma lista de elementos comparáveis e divide tal lista até que cada sublista possua tamanho unitário - trivialmente ordenável - e então combina as listas para obter a lista de entrada ordenada. Por mais poderosa que seja a ideia, a implementação do Merge Sort é bem simples e pode ser deduzida a partir dos pseudo-códigos 1 responsável pela combinação

das subsoluções e 2 responsável pelo processo de divisão.

---

**Algorithm 1:** MERGE

---

**Data:**  $\rho, \eta$  : *Sorted lists*  
**Result:** Sorted merge of  $\rho$  and  $\eta$   
**if**  $\rho = Nil$  or  $\eta = Nil$  **then**  
| **return**  $\rho + \eta$   
**else**  
| **if**  $car(\rho) \leq car(\eta)$  **then**  
| | **return**  $car(\rho) + MERGE(cdr(\rho), \eta)$   
| **else**  
| | **return**  $car(\eta) + MERGE(\rho, cdr(\eta))$

---

---

**Algorithm 2:** MERGESORT

---

**Data:**  $\tau$  : *List of comparable elements*  
**Result:** *Sorted permutation of  $\tau$*   
**if**  $length(\tau) \leq 1$  **then**  
| **return**  $\tau$   
**else**  
|  $prefix \leftarrow MERGESORT(first\_half(\tau));$   
|  $suffix \leftarrow MERGESORT(second\_half(\tau));$   
| **return**  $MERGE(prefix, suffix);$

---

## 4 Corretude do algoritmo Merge Sort

A corretude de um algoritmo passa por demonstrar que o mesmo possui certas características independente da instância e que essas características se verifiquem antes, durante e depois da execução. Para um algoritmo de ordenação, é esperado que o mesmo responda para qualquer entrada uma permutação ordenada da mesma, isto é, a saída não só deve estar ordenada como também o número de ocorrências de cada elemento deve ser o mesmo da lista de entrada.

A análise de MERGESORT nos leva a uma série de resultados intermediários que fortalecem a argumentação da corretude do algoritmo. Tais resultados estão expostos abaixo nesta seção e a formalização em PVS dos mesmos pode ser encontrado no repositório apresentado previamente na introdução deste texto.

**Lema 4.1.** *Para quaisquer  $\rho$  e  $\eta$ , listas, e  $n$  valor, o número de ocorrências de  $n$  em  $MERGE(\rho, \eta)$  é igual ao número de ocorrências de  $n$  em  $\rho$  mais o número de ocorrências de  $n$  em  $\eta$ .*

*Demonstração.* Fixando um  $n$  qualquer, induzindo sobre o tamanho das listas de entrada e valendo-se da definição de  $MERGE$ , essencialmente, chegamos a dois casos em que a propriedade descrita no lema deve valer. O primeiro caso é quando algumas das listas de entrada são nulas, neste caso, o algoritmo retorna uma das listas de entrada que, por si só, mantém as ocorrências de qualquer valor  $n$ . O segundo caso ocorre quando ambas as listas possuem ao menos um elemento, neste caso, em algum momento será inserido na lista resultante todas as ocorrências de  $n$  presentes nas listas de entradas.  $\square$

**Lema 4.2.** *Para quaisquer entradas  $\rho$  e  $\eta$ , listas, o tamanho da lista resultante de  $MERGE(\rho, \eta)$  é a soma dos tamanhos de  $\rho$  e  $\eta$ .*

*Demonstração.* Induzindo sobre o tamanho da listas e utilizando da definição de merge, a afirmação do lema deve ser verificado em três situações:

1.  $\rho$  ou  $\eta$  são listas nulas.
2. A cabeça de  $\rho$  é menor ou igual que a cabeça de  $\eta$ .
3. A cabeça de  $\rho$  é maior que a cabeça de  $\eta$ .

No primeiro caso, ao menos uma das listas tem tamanho zero, o algoritmo retorna então a concatenação de uma lista nula com outra lista  $\psi$ , que com certeza terá o mesmo tamanho que  $\psi$ . Para os outros dois casos basta demonstrar que tamanho de  $\rho$  mais tamanho de  $\eta$  é igual à  $1 + MERGE(cdr(\rho), \eta)$  ou  $1 + MERGE(\rho, cdr(\eta))$ .

A indução sobre o tamanho das listas neste lema nos garante que se a soma dos tamanhos de duas listas  $\psi$  e  $\gamma$  for estritamente menor que a soma dos tamanhos de  $\rho$  e  $\eta$ , então o tamanho de  $\psi$  mais tamanho de  $\gamma$  é igual ao tamanho de  $MERGE(\psi, \gamma)$ . Podemos utilizar o seguinte resultado para os pares de lista de entrada  $(cdr(\rho), \eta)$  e para  $(\rho, cdr(\eta))$  sem nenhum problema, uma vez que garantidamente nenhuma das listas é nula. Com isto, o problema é reduzido a verificação de

$$LENGTH(\rho) + LENGTH(\eta) = 1 + LENGTH(cdr(\rho)) + LENGTH(\eta)$$

e,

$$LENGTH(\rho) + LENGTH(\eta) = 1 + LENGTH(\rho) + LENGTH(cdr(\eta))$$

que são ambas, trivialmente, verdade.  $\square$

**Lema 4.3.** *Para quaisquer entradas  $\rho$  e  $\eta$ ,  $MERGE(\rho, \eta)$  é uma permutação de concanetação de  $\eta$  e  $\rho$ .*

*Demonstração.* Tomando  $\rho$  e  $\eta$  quaisquer, é sabido que a soma das ocorrências nas duas listas é igual às ocorrências na concanetação das mesmas. Logo, basta demonstrar que a soma das ocorrências em  $\rho$  e  $\eta$  é igual às ocorrências em  $MERGE(\rho, \eta)$ , o que efetivamente é o resultado verificado pelo lema 4.1.  $\square$

**Lema 4.4.** *Para quaisquer entradas  $\rho$  e  $\eta$ , se ambas as listas estão ordenadas então a resultante de  $MERGE(\rho, \eta)$  também será uma lista ordenada.*

*Demonstração.* Basta demonstrar que para qualquer iteração de  $MERGE$ , o menor elemento dentre ambas as listas é escolhido para inserção no final da lista parcialmente resultante. Uma vez que ambas as listas são ordenadas, necessariamente o menor elemento será a cabeça de  $\rho$  ou de  $\eta$ . Utilizando da definição de  $MERGE$  e induzindo sobre o tamanho de ambas as listas 3 situações emergem:

1. Alguma das listas  $\rho$  e  $\eta$  tem tamanho 0.
2. A cabeça de  $\rho$  é menor ou igual que a cabeça de  $\eta$ .
3. A cabeça de  $\rho$  é maior que a cabeça de  $\eta$ .

Na primeira situação, no máximo uma das listas possui elementos, então o elemento tomado para a composição da lista resultante será a cabeça (o menor elemento) daquela que ainda não é vazia, por conveniência, o algoritmo já retorna a lista inteira, mas se de fato tivesse que escolher apenas um, seria o menor disponível. O segundo e o terceiro caso são similares, a cabeça de  $\rho$  é igual ao menor elemento de  $\rho$  e o mesmo vale para a cabeça de  $\eta$ , logo, o menor entre os ambos será o menor dentre todos e este será escolhido para a inserção ao final da lista resultante, a chamada recursiva posterior cai novamente em algum dos 3 casos.  $\square$

**Lema 4.5.** *Para qualquer entrada  $\tau$ , lista, o tamanho de  $\tau$  é igual ao tamanho de  $MERGESORT(\tau)$ .*

*Demonstração.* Induzindo sobre o tamanho da lista  $\tau$  de entrada e valendo-se da definição de  $MERGESORT$ . Basta demonstrar que nos dois caminhos de execução do algoritmo não há modificação no tamanho da lista de entrada. O primeiro caso ocorre quando o tamanho da lista de entrada  $\tau$  é menor ou igual a um, que é trivial, uma vez que a resposta do algoritmo é a própria lista  $\tau$ .

No caso da chamada recursiva, o lema 4.2 nos garante que nenhuma chamada à  $MERGE$  altera o tamanho da lista resultante e a própria parada de  $MERGESORT$  conserva o tamanho da lista de entrada como visto anteriormente. Dessa forma,  $MERGESORT$  conserva o tamanho da lista de entrada na lista resultante.  $\square$

**Lema 4.6.** *Para qualquer lista  $\tau$ , a resultante de  $MERGESORT(\tau)$  é uma permutação de  $\tau$ .*

*Demonstração.* A definição de permutação de uma lista nos diz que para qualquer chave  $k$ , o número de ocorrências de  $k$  em  $\tau$  é o mesmo que em  $PERMUTATION(\tau)$ . Utilizando da definição de  $MERGESORT$  e aplicando indução sobre o tamanho da lista de entrada, obtemos dois casos em que faz necessário demonstrar que a saída do algoritmo é uma permutação da entrada. O primeiro caso, trivial, diz respeito à listas com tamanhos menores ou iguais a um, neste caso o algoritmo retorna a própria lista de entrada que por definição é uma permutação de si mesma.

O segundo caso a ser demonstrado é para listas de tamanho maior do que um, entradas as quais o algoritmo realiza ao menos uma chamada recursiva a si próprio. O lema 4.3 nos garante que a chamada à função  $MERGE$  efetivamente não altera o status de permutação de uma lista e o fato de o caso de parada da recursão de  $MERGESORT$  também não modificar esta configuração nos permite concluir que  $MERGESORT(\tau)$  é uma permutação de  $\tau$  para qualquer  $\tau$ .  $\square$

**Lema 4.7.** *Para qualquer lista  $\tau$ , a resultante de  $MERGESORT(\tau)$  é uma lista ordenada.*

*Demonstração.* Aplicando indução forte sobre o tamanho da lista  $\tau$  chegamos a dois casos a serem demonstrados. Se a lista tem tamanho menor ou igual a um, a mesma está trivialmente ordenada. Caso contrário, o lema 4.4



nos garante e o merge das chamadas recursivas de *MERGESORT* para cada uma das metades de  $\tau$ , *prefix* e *suffix*, tem como resultado uma lista ordenada desde que *MERGESORT(prefix)* e *MERGESORT(suffix)* sejam ambas listas ordenadas.

Aqui podemos nos valer de propriedade desta indução forte que diz que qualquer lista  $\phi$  tal que o tamanho da mesma seja estritamente menor que o tamanho de  $\tau$ , então *MERGESORT*( $\phi$ ) é uma lista ordenada. Sabemos que  $\tau$  tem tamanho maior ou igual que 2, logo qualquer metade desta lista terá tamanho estritamente menor que  $\tau$ .  $\square$

**Teorema 4.8.** *Para qualquer lista  $\tau$ , a resultante de *MERGESORT*( $\tau$ ) é uma permutação ordenada de  $\tau$ .*

*Demonstração.* Aplicando os lemas 4.7 e 4.6, a propriedade da corretude torna-se evidente.  $\square$

## 5 Análise de complexidade do Merge Sort

A complexidade de um algoritmo de ordenação se baseia em contar o número de comparações realizadas durante sua execução. Neste sentido, existe uma classificação de algoritmos de ordenação chamada algoritmos de ordenação por comparação, que, segundo Cormen [1], são algoritmos baseados em comparações entre dois elementos e efetuam pelo menos  $\Omega(n \lg n)$  comparações no pior caso.

O Merge Sort é um algoritmo de ordenação por comparação, e portanto, sua análise de complexidade se baseia em contar quantas comparações são efetuadas para ordenar uma lista de tamanho  $n$ . Para tanto, foram necessárias modificações no algoritmo, de forma a incluir um contador de comparações. O pseudo-código abaixo representa a ideia da inclusão de um contador no Merge Sort:

---

**Algorithm 3: COUNT-MERGE**

---

**Data:**  $\rho, \eta$  : Sorted lists,  $\sigma$  : natural

**Result:** Sorted merge of  $\rho$  and  $\eta$ , number of comparisons

**if**  $\rho = Nil$  or  $\eta = Nil$  **then**

**return**  $\rho + \eta, \sigma$

**else**

**if**  $car(\rho) \leq car(\eta)$  **then**

**return**  $car(\rho) + MERGE(cdr(\rho), \eta, \sigma + 1)$

**else**

**return**  $car(\eta) + MERGE(\rho, cdr(\eta), \sigma + 1)$

---

---

**Algorithm 4: COUNT-MERGESORT**

---

**Data:**  $\tau$  : List of comparable elements

**Result:** Sorted permutation of  $\tau$ , number of comparisons

**if**  $length(\tau) \leq 1$  **then**

**return**  $\tau, 0$

**else**

$prefix, \sigma \leftarrow MERGESORT(first\_half(\tau));$

$suffix, \iota \leftarrow MERGESORT(second\_half(\tau));$

**return**  $MERGE(prefix, suffix, \sigma + \iota);$

---

A adição de um contador no Merge Sort permite a análise de comparações, porém, não é possível assegurar que o contador não inseriu um erro no algoritmo. Portanto, é necessário provar os seguintes lemas, que asseguram a equivalência entre o Merge Sort com e sem contador:

**Lema 5.1.**  $\forall \rho, \eta$  listas de naturais, a resultante de  $MERGE(\rho, \eta)$  é igual à resultante de  $COUNT - MERGE(\rho, \eta, \sigma)$ , a não ser por  $\sigma$ .

*Demonstração.* A ideia da prova é mostrar que a lista retornada por  $MERGE$  é igual à lista retornada por  $COUNT - MERGE$ . Utilizando indução forte sobre a soma tamanho das listas  $\rho$  e  $\eta$ , temos que para quaisquer listas cujo a soma dos tamanhos das mesmas é menor que a soma do tamanho das listas, a propriedade vale. Portanto basta expandir a definição de  $MERGE$  e  $COUNT - MERGE$  e analisar os seguintes casos, no momento da chamada recursiva:

1. cabeça de  $\rho$  é maior ou igual que a cabeça de  $\eta$
2. a cabeça de  $\rho$  é menor que a cabeça de  $\eta$

Em ambos, temos uma construção de uma lista no qual podemos ver que a propriedade da indução pode ser aplicada, pois temos que tamanho da cauda de  $\rho$  + tamanho de  $\eta$  é menor que tamanho de  $\rho$  + tamanho de  $\eta$ , e precisamos mostrar que inserir a cabeça de  $\rho$  na lista construída resulta na mesma lista para as duas funções. De fato, ao aplicar a hipótese de indução, temos que inserir o mesmo elemento em duas listas iguais resulta em duas listas iguais, o que é trivialmente verdade.  $\square$

**Lema 5.2.**  $\forall \rho$ , lista de naturais, a resultante de  $MERGESORT(\rho)$  é igual à resultante de  $COUNT - MERGESORT(\rho)$ , a não ser pelo contador retornado.

*Demonstração.* De forma semelhante ao lema anterior, a ideia da prova é mostrar que o retorno de  $MERGESORT$  e  $COUNT - MERGESORT$  são equivalentes. Com indução forte, temos que para quaisquer listas cujo a soma dos tamanhos das mesmas é menor que a soma do tamanho das listas, a propriedade vale. Com isso, aplicamos a hipótese de indução para **prefix** e **suffix** e expandimos as definições de  $MERGESORT$  e  $COUNT - MERGESORT$ . Basta aplicar o lema anterior para  $MERGE$  e  $COUNT - MERGE$  na expansão de  $MERGESORT$  e  $COUNT - MERGESORT$ , e o seguinte é provado.  $\square$

O lema a seguir é um lema auxiliar, necessário para a prova do lema 5.6.

**Lema 5.3.**  $\forall \rho$ , lista de naturais, o tamanho da resultante de  $COUNT - MERGESORT$  é igual ao tamanho de  $\rho$ .

*Demonstração.* Para tal, basta utilizar o lema 5.2 e o lema 4.5.  $\square$

Os lemas seguintes são acerca da complexidade do Merge Sort, utilizando os contadores de comparações.

**Lema 5.4.**  $\forall \rho, \eta$ , listas de naturais, o valor do contador de comparações retornado pela chamada  $COUNT - MERGE(\rho, \eta, 0)$  é menor ou igual que  $\mathbf{n} + \mathbf{m}$ , onde  $\mathbf{n}$  é o tamanho de  $\rho$  e  $\mathbf{m}$  é o tamanho de  $\eta$

*Demonstração.*  $\square$

**Lema 5.5.**  $\forall \rho, \eta$ , listas de naturais, e  $\sigma$ , um natural, o valor do contador de comparações retornado pela chamada  $COUNT-MERGE(\rho, \eta, \sigma)$  é menor ou igual que  $\mathbf{n} + \mathbf{m} + \mathbf{c}$ , onde  $\mathbf{n}$  é o tamanho de  $\rho$ ,  $\mathbf{m}$  é o tamanho de  $\eta$  e  $\mathbf{c}$  é o número de comparações realizadas anteriormente, ou seja,  $\sigma$ .

*Demonstração.* □

**Lema 5.6.**  $\forall \rho$ , lista de naturais, tal que tamanho de  $\rho$  é maior que zero, o valor do contador de comparações retornado pela chamada  $COUNT-MERGESORT(\rho)$  é menor ou igual a  $\mathbf{n} + \mathbf{n} \log \mathbf{n}$ , onde  $\mathbf{n}$  é o tamanho de  $\rho$ .

*Demonstração.* □

## 6 Conclusão

O trabalho desenvolvido foi desafiador no sentido de ter que se preocupar com pequenos detalhes de cada prova. Os lemas e as provas apresentados acima apesar de simples e por vezes evidentes levaram a formalizações complexas e de difícil entendimento à olho nu, por vezes, uma simples enunciação correta mas não favorável ao longo do trabalho tornou o desenvolvimento da prova impraticável. Isso traduziu-se em alguns lemas sem uma prova completa ainda que a argumentação tenha oferecido uma boa base para a veracidade dos mesmos.

De fato tanto o PVS quanto outros assistentes de prova não são indicados para quem deseja aprender a fazer uma prova, e sim para aqueles que já dominam a prova e não possuem algum porém em relação à fundamentação lógica por trás. A conjunto de pessoas que utilizam assistente de provas ainda é muito restrito mas há um movimento para que isto cresça atrelado com a evolução deste tipo de Software.

Em relação ao trabalho aqui apresentado, é deixada uma formalização completa da corretude do Merge Sort e um largo passo em relação à formalização da complexidade do algoritmo em seu pior caso, ambas em PVS. As provas deixadas estão hospedadas em modo público no repositório citado acima e podem ser verificadas e modificadas à vontade. Vale salientar que o uso da Nasa PVS Library ou similar é imprescindível para a formalização da complexidade uma vez que abordagem aqui adotada utiliza de propriedades não triviais da função logarítmica.

## Referências

- [1] CORMEN, T., LEISERSON, C., AND STEIN, R. *Algoritmos: teoria e prática*. ELSEVIER EDITORA, 2012.