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

5 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 sequentes. O PVS permite também que uma prova possa ser revisitada em uma representação gráfica que revela cada passo bem como suas depedências.

O objetivo deste trabalho é analisar assintóticamente 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`.

Esse documento se organizará, daqui em diante, por um capítulo 2 de revisão teórica. Em seguida, no capitulo 3, a apresentação do algoritmo Merge Sort. O capitulo 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 capitulo 4 apresentará, como o capitulo anterior, a ideia abordada e sua argumentação. Por fim, o capitulo 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 capitulo 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 execucação de algoritmos. Segue-se então as definições:

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

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

A definição de Ω é similar a definição de O, com a diferença que Ω 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 Θ 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 cauda, 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: ρ, η : *Sorted lists*
Result: Sorted merge of ρ and η
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: τ : *List of comparable elements*
Result: *Sorted permutation of* τ
if $length(\tau) \leq 1$ **then**
  return τ
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 ρ e η, listas, e n valor, o número de ocorrências de n em $MERGE(\rho,\ \eta)$ é igual ao número de ocorrências de n em ρ mais o número de ocorrências de n em η.*

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. □

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

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

1. ρ ou η são listas nulas.
2. A cabeça de ρ é menor ou igual que a cabeça de η.
3. A cabeça de ρ é maior que a cabeça de η.

No primeiro caso, ao menos uma das listas tem tamanho zero, o algoritmo retorna então a concanetação de uma lista nula com outra lista ψ, que com certeza terá o mesmo tamanho que ψ. Para os outros dois casos basta demonstrar que tamanho de ρ mais tamanho de η 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 ψ e γ for estritamente menor que a soma dos tamanhos de ρ e η, então o tamanho de ψ mais tamanho de γ 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. □

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

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

Lema 4.4. *Para quaisquer entradas ρ e η, 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 elementos 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 ρ ou de η. Utilizando da definição de $MERGE$ e induzindo sobre o tamanho de ambas as listas 3 situações emergem:

1. Alguma das listas ρ e η tem tamanho 0.

2. A cabeça de ρ é menor ou igual que a cabeça de η.

3. A cabeça de ρ é maior que a cabeça de η.

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 ρ é igual ao menor elemento de ρ e o mesmo vale para a cabeça de η, 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.

□

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

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

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. □

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

Demonstração. A definição de permutação de uma lista nos diz que para qualquer chave k, o número de ocorrências de k em τ 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 à 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 sí 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 τ para qualquer τ. □

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

Demonstração. Aplicando indução forte sobre o tamanho da lista τ chegamos a dois casos a serem demonstrados. Se a lista tem tamanho menor ou igual à 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 τ, $prefix\ e\ suffix$, 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 ϕ tal que o tamanho da mesma seja estritamente menor que o tamanho de τ, então $MERGESORT(\phi)$ é uma lista ordenada. Sabemos que τ tem tamanho maior ou igual que 2, logo qualquer metade desta lista terá tamanho estritamente menor que τ. □

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

Demonstração. Aplicando os lemas 4.7 e 4.6, a propriedade da corretude torna-se evidente. □

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 $\Omega(\boldsymbol{n\ lg\ n})$ 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 $\boldsymbol{n}$. 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: ρ, η : *Sorted lists*, σ : *natural*
Result: *Sorted merge of* ρ *and* η, *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: τ : *List of comparable elements*
Result: *Sorted permutation of* τ, *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. *$\forall$ ρ, η listas de naturais, a resultante de $MERGE(\rho,\ \eta)$ é igual à resultante de $COUNT - MERGE(\rho,\ \eta, \sigma)$, a não ser por σ.*

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 ρ e η, 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 ρ é maior ou igual que a cabeça de η

2. a cabeça de ρ é menor que a cabeça de η

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 ρ + tamanho de η é menor que tamanho de ρ + tamanho de η, e precisamos mostrar que inserir a cabeça de ρ 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. *$\forall$ ρ, 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 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 sequente é provado. $\square$

O lema a seguir é um lema auxiliar, necessário para a prova do lema 5.7.

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

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. Os lemas 5.4 e 5.5 são acerca da complexidade da rotina $MERGE$. Já o lema 5.7 busca mostrar a complexidade de $MERGESORT$ por completo.

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 $\boldsymbol{n} + \boldsymbol{m}$, onde $\boldsymbol{n}$ é o tamanho de ρ e $\boldsymbol{m}$ é o tamanho de η*

Demonstração. Primeiramente, utilizamos indução forte no tamanho das listas ρ e η, e temos, como hipótese de indução, que a propriedade do lema vale para quaisquer listas tal que a soma dos tamanhos dessas listas é menor que a soma dos tamanhos de ρ e η. Então, basta expandir $COUNT-MERGE$ e analisar cada um dos casos:

1. Temos que provar que a soma dos tamanhos de ρ e η é maior ou igual que 0, dado que ρ é nulo ou η é nulo.

2. Temos que provar que $1 + \sigma$ é menor ou igual a tamanho de ρ + tamanho de η, onde σ é o contador retornado pela chamada $COUNT-MERGE(cdr(\rho),\ \eta,\ 0)$.

3. Temos que provar que $1 + \sigma$ é menor ou igual a tamanho de ρ + tamanho de η, onde σ é o contador retornado pela chamada $COUNT-MERGE(\rho,\ cdr(\eta),\ 0)$.

Para o item 1, basta verificar que o tamanho não pode ser negativo, então é sempre maior ou igual a zero.

Para o item 2, utilizamos a hipótese de indução para tamanho de $cdr(\rho)$ + tamanho de η e expandimos a definição de tamanho de listas não-nulas para verificar que a propriedade é verificada.

Para o item 3, basta utilizar a mesma ideia da prova do item 2, com a hipótese de indução sendo utilizada para $cdr(\eta)$. $\square$

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

Demonstração. Por ser uma generalização do lema anterior, a prova desse lema se dá de forma bem semelhante. Começamos utilizando indução forte em $LENGTH(\rho) + LENGTH(\eta)$ e temos como hipótese que a propriedade vale para quaisquer listas tal que a soma dos tamanhos dessas listas é menor que a soma dos tamanhos de ρ e η. Ao expandir a definição de $COUNT-MERGE$ temos os casos:

1. σ é menor ou igual a $LENGTH(\rho) + LENGTH(\eta) + \sigma$, onde σ é o contador retornado pela chamada $COUNT-MERGE(\rho,\ \eta,\ \sigma)$.

2. $1 + \sigma$ é menor ou igual a $LENGTH(\rho) + LENGTH(\eta) + \sigma$, onde σ é o contador retornado pela chamada $COUNT{-}MERGE(cdr(\rho),\ \eta,\ \sigma)$.

3. $1 + \sigma$ é menor ou igual a $LENGTH(\rho) + LENGTH(\eta) + \sigma$, onde σ é o contador retornado pela chamada $COUNT{-}MERGE(\rho,\ cdr(\eta),\ \sigma)$.

Para o item 1, assim como no lema anterior, temos que o tamanho de uma lista não pode ser negativo, portanto, o sequente é trivialmente verdade.

Para o item 2, utilizamos a hipótese de indução para $cdr(\rho)$ e expandirmos a definição de LENGTH, que mostra que o sequente é verdade.

Para o item 3, basta utilizar a ideia da prova do item 2, porém utilizando a hipótese de indução para $cdr(\eta)$. $\square$

O lema asseguir representa a complexidade de fato do algoritmo, e utiliza o seguinte axioma, também utilizado por Cormen[1]:

Axioma 5.6. *$\forall\rho$, lista de naturais, floor(LENGTH(ρ)/2) = LENGTH(ρ)/2, onde **floor** denota o arrendondamento para baixo.*

Lema 5.7. *$\forall\rho$, lista de naturais, tal que tamanho de ρ é maior que zero, o valor do contador de comparações retornado pela chamada $COUNT - MERGESORT(\rho)$ é menor ou igual a **n + n log n**, onde **n** é o tamanho de ρ.*

Demonstração. A ideia central da prova é utilizar substituições para chegar no resultado final. Para tanto, começamos a prova utilizando indução forte sobre LENGTH(ρ), sendo a hipótese de indução que a propriedade vale para listas cujo tamanho é menor que o tamanho de ρ, e expandimos a definição de $COUNT - MERGESORT$. Usamos o lema 5.5 e o lema 5.3 para chegar ao sequente: o contador retornado por $COUNT - MERGE$ aplicada a $COUNT - MERGESORT$ expandido é LENGTH(ρ) + σ + ι, onde σ é o valor do contador de $COUNT - MERGESORT$ aplicado ao prefixo de ρ e ι é o valor do contador de $COUNT - MERGESORT$ aplicado ao sufixo de ρ, ou seja, chegamos à equação de recorrência $\mathrm{T(n)} \leq \mathrm{T}(\lceil\frac{n}{2}\rceil) + \mathrm{T}(\lfloor\frac{n}{2}\rfloor) + \mathrm{n}$, onde **n** é LENGTH($\rho$).

Com a equação de recorrência presente no sequente, utilizamos a hipótese de indução para suffix e prefix, e o axioma 5.6, que simplifica a recorrência e a transforma em $\mathrm{T(n)} \leq 2 * \mathrm{T}(\frac{n}{2}) + \mathrm{n}$. Basta utilizar então as propriedades de divisão e inverso do logaritmo para provar o sequente. $\square$

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 dificil 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 imprencidível 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.