Universidade de Brasília
Instituto de Ciências Exatas

Departamento de Ciência da Computação

117366 - Lógica Computacional 1 Turma: A

# Prova de propriedades de algoritmos de ordenação com uso da ferramenta de prova PVS

Isabelle Alex dos S. B. Caldas - 17/0105636
Oscar E. B. Madureira da Silva - 17/0112209
Thiaggo Ferreira Bispo de Souza - 17/0157024

18 de novembro de 2019

## 1 Introdução

Algoritmos de ordenação possuem várias aplicações em ciência da computação. Dois algoritmos de ordenação bem conhecidos são o merge sort e radix sort: o merge sort é um algoritmo estável que se baseia no paradigma de dividir para conquistar, ou seja, divide o problema dado como uma lista/array em metades diversas vezes até a sua forma atômica (uma lista/array de tamanho um), e então utiliza a função merge que, por sua vez, une duas listas já ordenadas e uma única lista ordenada; o radix sort também é estável, porém, ele ordena de maneira diferente. Inicialmente ele ordenada as entradas pelos dígitos menos significativos e continua ordenando em direção ao dígito mais significativo de forma que no final as entradas encontram-se completamente ordenadas.

# 2 Especificação do problema e explicação do método de solução

Nesse projeto devemos provar que radix sort é uma função de ordenação, ou seja, dada uma lista l, retorna uma lista ordenada lexicograficamente de acordo com as pré-ordens <<e <= e para isso devemos provar também conjecturas que irão nos auxiliar nessa prova, como provar que merge sort e radix sort de uma lista l são permutações dessa lista.

Começamos com a prova de que merge sort de uma lista l é uma permutação dessa lista, para isso usamos indução forte em length(l), pois a definição de merge sort é baseada nessa função e é recursiva. Em seguida, partimos para uma prova semelhante, porém agora com radix sort. Para isso, utilizamos a mesma ideia da primeira prova, uma vez que a definição de radix sort é fundamentada na definição de merge sort que, como dito antes, é fundamentada em length(l), ou seja, utilizamos como hipótese a conjectura de que merge sort de uma lista l é uma permutação de l. Para a questão final, nos baseamos apenas em definições e em lemas auxiliares.

# 3 Explicação das soluções

## 3.1 Questão 1 - merge sort is permutation

Partindo da indução forte descrita acima, temos de provar que o merge sort(x) é uma permutação de x, tendo como hipótese que o merge sort de uma lista y é uma permutação de y, para qualquer y que tenha o tamanho menor que x. Em seguida, utilizamos a definição do merge sort, que nos leva a dois casos: O primeiro em que o tamanho da lista x é menor ou igual a 1, o que é trivial, pois assumindo que a lista tem um elemento a permutação dela também terá um elemento, agora assumindo que a lista é vazia, pela definição de permutação, o número de ocorrência de x qualquer é sempre igual a zero; E o segundo em que o oposto ocorre, o tamanho da lista x é maior que 1.

Para o segundo caso temos que provar que o merge dos merge sorts das metades da lista x é uma permutação de x e para isso usamos a definição de permutações que diz que o número de ocorrências de qualquer número em merge dos merge sorts das metades da lista é igual ao número de ocorrências de um número qualquer em x para isso usamos o lema merge occurrence, que diz que o número de ocorrências de um elemento x em um lista formada pelo merge de duas listas l1 e l2 é igual à soma do número de ocorrências de x na primeira lista (l1) com o número de ocorrências de x na segunda lista (l2), assim, como os merge sorts das metades das listas também resulta em listas, foi possível substituir o consequente, que agora afirma que a soma das ocorrências de x1 nos merge sorts das metades das listas é igual ao numero de ocorrências de x1 na lista x. Para resolver isso, utilizamos a hipótese de indução, em que, se aplicada a definição de permutações, afirma que se o tamanho de uma lista y for menor que o tamanho de uma lista x, então o número de ocorrências de um elemento x no merge sort de uma lista y é igual ao numero de ocorrências desse mesmo elemento x na lista. Assim, dividindo-a em dois casos e considerando as metades das listas, temos o caso em que o tamanho da metade da lista x é maior que o tamanho da lista x - o que é obviamente falso - e temos o caso em que o tamanho da metade da lista x é menor que o tamanho da lista x e logo o número de ocorrências de um elemento x1 no merge sort de uma metade da lista x é igual ao número de ocorrências de x1 na metade da lista x.

Dessa forma, podemos substituir esse antecedente no consequente e o que temos que provar agora é que o número ocorrências de x1 numa lista x é igual ao número de ocorrências de x1 na primeira metade da lista somado ao número de ocorrências de x1 na segunda metade da lista. Assim, podemos usar o lema merge ocurrence, que afirma que o número de ocorrências de um elemento x1 em um merge de duas listas l1 e l2 é igual à soma das ocorrências de x1 em l1 e de x1 em l2, e substituir no consequente que afirmará que o número de ocorrências de um elemento x1 no merge das metades das listas é igual ao número de ocorrências de x1 em uma lista x. Neste momento, buscamos o lema merge is permutation, que afirma que o append de de duas listas l1 e l2 é uma permutação do merge dessas mesmas listas, com isso, podemos aplicar a definição de permutations mudando o consequente para "o número de ocorrências de x1 no append das metades de uma lista x é igual ao número de ocorrências de x1 na lista x- o que é verdadeiro pois a junção da primeira metade de uma lista com a segunda metade da lista é a própria lista

## 3.2 Questão 2 - radixsort permutes

Para essa questão devemos provar que o radix sort de uma lista l é uma permutação dessa lista. Para resolvermos essa questão recorremos à definição de radix sort, que afirma que o radix sort de uma lista é igual ao merge sort de ordem << do merge sort de ordem <= de uma lista l, assim, o nosso consequente agora afirma que esse merge sort de um merge sort de l é uma permutação de l. Podemos, então, nos valer da questão anterior - que afirma que o merge sort de uma lista é uma permutação da mesma - e utilizá-la como um lema aplicado à ordem << de forma que possamos aproveitar a definição de permutations e transformar o antecedente para “as ocorrências de x no merge sort de ordem << do merge sort de ordem <= da lista l são iguais às ocorrências de x no merge sort de ordem <= da lista l” e o consequente para “as ocorrências de x no merge sort de ordem <<do merge sort de ordem <= da lista l são iguais às ocorrências de x na lista l”.

O próximo passo é utilizar o que temos no antecedente no consequente, tornando-o “ as ocorrências de x no merge sort de ordem <= da lista l são iguais às ocorrências de x em l” e repetir o mesmo processo feito anteriormente, mas para a ordem <=, ou seja, utilizamos o lema aplicado anteriormente empregado à ordem <= e usamos a definição de permutations. Dessa forma, temos o antecedente como “as ocorrências de x no merge sort de ordem <= uma lista l são iguais às ocorrências de x em l” o que é igual ao consequente, concluindo a prova.

## 3.3 Questão 3 - radixsort sorts

Dada uma lista “l” temos que provar que a função radix sort de uma lista “l” retorna uma lista ordenada na ordem lexicográfica. Primeiramente, usaremos a definição de radix sort (pois, como sua definição é baseada no merge sort, então, há um número maior de lemas para nos auxiliar futuramente na prova), denotaremos merge sort para a ordem <<por “merge sort mm” e merge sort para a ordem <= por “merge sort mi”, assim é necessário provar que o “merge sort mm” do “merge sort mi” da lista “l” retorna uma lista ordenada. E, em seguida, para aplicarmos a definição de is sorted? dizemos que k é o k-ésimo, menor que o tamanho da lista elemento na lista, assim, basta provar que o k-ésimo elemento do merge sort do merge sort é lexicograficamente menor que que seu sucessor. Enfim, usaremos a definição de lexicograficamente menor, assim basta provar ao menos uma coisa:

- Que o k-ésimo elemento do merge sort do merge sort de l é <<que o seu sucessor, e que seu sucessor não é <<do que ele

- Ou que k-ésimo elemento do merge sort do merge sort de é <<que o seu sucessor, que o seu sucessor é <<do que ele, e que o k-ésimo elemento do merge sort do merge sort de é <= que o seu sucessor

Agora, como temos que provar a ordem <<para o k-ésimo em ambos os casos, podemos fazer essa afirmação verdadeira adicionando o lema merge is sorted de ordem <<e para a lista "merge sort mi", agora é suficiente provar que:

- Que o k+1-ésimo elemento do merge sort do merge sort de "l" não é <<que o k-ésimo elemento

- Ou que o k+1-ésimo elemento do merge sort do merge sort de "l" é <<que o k-ésimo elemento e que o k-ésimo elemento do merge sort do merge sort de é <= que o seu sucessor

Assim suponhamos que a afirmação "k+1-ésimo elemento do merge sort do merge sort de "l" é <<que o k-ésimo elemento" seja falsa. Para esse caso, a prova estaria completa, uma vez que um dos consequentes foi satisfeito; quando ela for verdadeira, devemos provar, ainda, que o k-ésimo elemento do merge sort do merge sort de é <= que o seu sucessor.

Dessa forma, como o consequente possui duas partes e a segunda parte é a única que depende de mais uma afirmação, podemos assumir que o k+1-ésimo elemento do merge sort do merge sort de "l" é <<que o k-ésimo elemento, assim, nos resta provar a partir dos nossos antecedentes que o k-ésimo elemento do merge sort do merge sort de é <= que o seu sucessor.

Para isso, com o intuito de simplificar a proposição, utilizamos o lema merge sort is conservative para retirar o merge sort mais externo e manter a ordem relativa dos elementos, uma vez que esse lema afirma que, para elementos iguais, a ordem original será mantida, ou seja, para k e k+1, existe i e j tais que i<.j, o k-ésimo elemento do merge sort do merge sort da lista "l" é igual i-ésimo elemento do "merge sort mi" de "l" e o k+1-ésimo elemento do do merge sort do merge sort da lista "l" é igual ao j-ésimo elemento do "merge sort mi", assim, podemos, substituir o nosso consequente por esses resultados.

Para avançarmos nessa prova, foi necessário usar o lema is sorted implies monotone para a ordem <= que afirma que se temos uma lista "l" ordenada então para todo j menor que o tamanho de l e todo i menor que j, o i-ésimo elemento de l é <= ao j-ésimo elemento de l. Usamos então o lema merge sort is sorted para a ordem <= de forma que, se substituirmos a lista "l" do lema is sorted implies monotone por "merge sort mi" de l, satisfazemos a condição da implicação e adicionamos o seu consequente ao nosso conjunto de hipóteses, ou seja, adicionamos "para todo j menor que o tamanho de l e todo i menor que j, o i-ésimo elemento de l é <= ao j-ésimo elemento de l" ao nosso antecedente. Dessa forma, se substituirmos l por "merge sort mi", teremos a mesma coisa no antecedente e no consequente, ou seja, concluímos a prova.

### 3.4 Correção da solução proposta

## 4 Conclusão

O intuito desse projeto é familiarizar os alunos com o PVS, uma ferramenta de prova lógica com diversos mecanismos de simplificação onde podemos realizar provas mais complexas, e utilizar os conceitos teóricos ensinados em sala de aula. O PVS nos ajudou a fixar as teorias estudadas durante o semestre, pondo-as em prática ao provar o funcionamento de algoritmos de ordenação.

Foi possível observar conceitos do Cálculo de Sequentes como a regra da implicação à direita e a c-equivalência ao longo do projeto, o que mostra que as simplificações feitas pela ferramenta não nos distancia da teoria estudada durante o semestre. Ao provarmos propriedades de algoritmos como merge sort e radix sort, que possuem diversas utilidades na área da Computação, foi possível notar, também, que ferramentas de prova lógica são poderosas aliadas para a certificação de um *software* pois esses algoritmos possuem definições recursivas e teste criado poderia comprovar o seu funcionamento para todas as possíveis entradas.