Universidade de Brasília
Departamento Ciência da Computação
RadixSort 1/2019

Relatório Final

Alunos:
Thiago Santana Marques 14/0164049
Flávio Amaral e Silva 13/0110728
Geraldino Antonio da Silva 13/0112267
Professor: Flávio L. C. de Moura

junho
2019

1 Introdução e contextualização do problema

A origem da palavra algoritmo é divergente entre pesquisadores, a mais difundida é a de Mohamed ben Musa Al-Khwarizmi, um matemático persa do século IX que teve suas obras traduzidas para o ocidente e uma destas recebeu o nome Algorithmi de numero indorum (indiano), acerca dos algoritmos que trabalham sobre o sistema de numeração decimal. Porém, para a Ciência da Computação o conceito de algoritmo foi formalizado em 1936 por Alan Turing (Máquina de Turing) e Alonzo Church, que formaram as primeiras fundações da Ciência da computação. Sendo esta formalização descrita a seguir: *Um algoritmo é um conjunto não ambíguo e ordenado de passos executáveis que definem um processo finito.*

Em 1890, foi desenvolvido uma máquina por Hermann Hollerith que fazia a leitura de cartões de papel perfurados em código BCD (BinaryCoded Decimal). A informação perfurada no cartão era lida em uma tabuladora que dispunha de uma estação de leitura equipada com uma espécie de pente metálico em que cada dente estava conectado a um circuito elétrico. Antes da utilização da máquina de cartões perfurados, o censo nos Estados Unidos era realizado em 10 anos, o que prejudicava o bom andamento das políticas públicas que precisavam dos dados dos censos para serem melhor planejadas. Após a máquina de Herman Hollerith o censo passou a ser realizado em 1 ano. O RadixSort nasceu com a necessidade de ordenar cartões perfurados a partir de uma chave única processada por partes. Em resumo, a estratégia do RadixSort: O algoritmo realiza p iterações, onde p é o número de dígitos. Por fim, utiliza dois vetores: vOcor, que corresponde ao número de ocorrências de elementos, e vTemp, que é um vetor temporário que mantém os elementos ordenados por um certo dígito.

1. Inicializa $vOcor$;
2. O vetor $vOcor$ recebe o número de ocorrências de cada $i-ésimo$ dígito;
3. Uma vez preenchido o $vOcor$, são contabilizados nele os offsets para cada dígito (posições onde iniciam os elementos que possuem certo valor de dígito);
4. Com base nesses offsets, $vTemp$ recebe os elementos do vetor ordenado pelo $i - ésimo$dígito;
5. Transfere-se os elementos de $vTemp$ para vetor;

A indução é uma poderosa técnica de prova que permite estender soluções de subproblemas menores para subproblemas maiores. O benefício imediato dessa metodologia de construção de algoritmos é que a corretude do algoritmo é obtida diretamente.

Portanto, queremos provar por indução matemática que o processo de ordenação do RadixSort funciona utilizando um algoritmo estável como algoritmo auxiliar. Segundo Rezende a técnica de demonstrações por indução tem especial interesse em computação pois possui a característica de ser construtiva, evidenciando os passos necessários para se obter o objeto tese do teorema.

Neste projeto, utilizamos o assistente de demonstração PVS. O sistema oferece um ambiente mecanizado para especificação e verificação formal, é amplamente utilizado para formalização de conceitos matemáticos e provas em áreas como análise, teoria dos grafos e teoria dos números; na verificação de hardware, algoritmos sequenciais e distribuídos; e como uma ferramenta de verificação de back-end para sistemas de álgebra computacional e verificação de código. No PVS temos as especificações como sendo arquivos de texto ASCII salvos com a extensão .pvs. Logicamente são organizadas e modularizadas em teorias (parametrizadas ou não), permitindo generalização e reusabilidade. A sintaxe de uma teoria é dada por:

```
Id [TheoryFormals]: THEORY
  [Exporting]
BEGIN
  [AssumingPart]
  [TheoryPart]
END Id
```

Especificações em PVS são fortemente tipadas, significando que toda expressão deve ter um tipo associado:

1. Tipos Pré-definidos
2. Tipos Não Interpretados
3. Subtipos Predicados
4. Tipo Enumeração
5. Tipo n-Tupla
6. Tipo Registro
7. Tipo Dependente
8. Tipo Função

A declaração básica de uma variável é seguido por seu nome e tipo. Um exemplo é:

n: VAR nat

Constantes podem ser introduzidas especificando seus tipos e opcionalmente seus valores.

$$g(x : int) : int = x + 1.$$

Declarar uma função recursiva no PVS implica em definir o valor para todo o domínio. Utilizamos o comando MEASURE para especificar uma medida através de uma função que justifica sua boa-formação (toda função recursiva tem que terminar). Um exemplo de definição recursiva de fatorial:

```
fatorial(x: nat): RECURSIVE nat =
        IF x = 0 THEN 1 ELSE x * fatorial (x - 1) ENDIF
        MEASURE (LAMBDA (x : nat): x)
```

Declarar uma função implica em introduzir axiomas, assunções (restrições), obrigações (que são geradas pelo sistema por meio dos TCCs e não podem ser especificadas pelo usuário) e teoremas - seguidos por palavras reservadas CHALLENGE, CLAIM, CONJECTURE, COROLLARY, FACT, FORMULA, LAW, LEMMA, PROPOSITION, SUBLEMMA ou THEOREM, que possuem a mesma semântica para o PVS, porém permitem uma maior diversidade na classificação das fórmulas pelo usuário. [Referenciar]

As expressões no PVS podem aparecer no corpo de uma fórmula ou na declaração de constantes, bem como em predicados de subtipo ou nos parâmetros atuais da instância de uma teoria. Temos:

1. Expressões booleanas
2. Expressões Numéricas
3. Expressões Condicionais
4. Abstração Lambda
5. Expressões LET e WHERE

Por fim, o PVS possui um provador acoplado, que é uma coleção de procedimentos de inferência que são aplicados de forma interativa sob a orientação do usuário dentro de uma estrutura de Cálculo de Sequentes.

2 Explicação das soluções

Questão 01:

```
Rule? (measure-induct+ "k-d" ("k" "d"))
Inducting on measure: k - d,
 with variables: (k d),
this simplifies to:
radix_sort_d_sort :

{-1}  FORALL (y_1: nat, y_2: {d: nat | d <= y_1}):
        FORALL (l: list[nat]):
          y_1 - y_2 < x!1 - x!2 IMPLIES
           is_sorted_ud?(l, y_2) =>
            is_sorted_ud?(radixsort(l, y_1, y_2), y_1 + 1)
  |-------
{1}   FORALL (l: list[nat]):
        is_sorted_ud?(l, x!2) =>
         is_sorted_ud?(radixsort(l, x!1, x!2), x!1 + 1)
```

Começamos a prova utilizando a indução no consequente com o comando measure-induct+.

Trocamos os nomes das variáveis x!1 e x!2 por k e d

Após eliminar o quantificador universal FORALL com o comando skeep, geramos duas subprovas ao instanciar o sequente -1.

Aplicamos a definição de "**radixsort**"no **sequente 1**.

Chegamos no caso em que $d = k$. Precisamos separar os casos com o comando **lift-if**.

Quebramos o caso novamente com o comando **split**

Podemos provar cada um separadamente, utilizamos o comando **flatten** para simplificar a disjunção e por fim podemos carregar o **lemma** *"mergesortd-sorts"*.

Queremos provar que a lista está ordenada até $k + 1$ e o **lemma** carregado nos trás esta definição, basta instanciar o **sequente -1** para fechar a prova.

Para subárvore da direita, provamos o caso em que k e d são diferentes. Quebramos o sequente novamente com o comando **split** e obtemos uma subárvore a direita.

O comando **typepred** fez a avaliação automática em "d".

Expandimos d e k para fechar a prova.

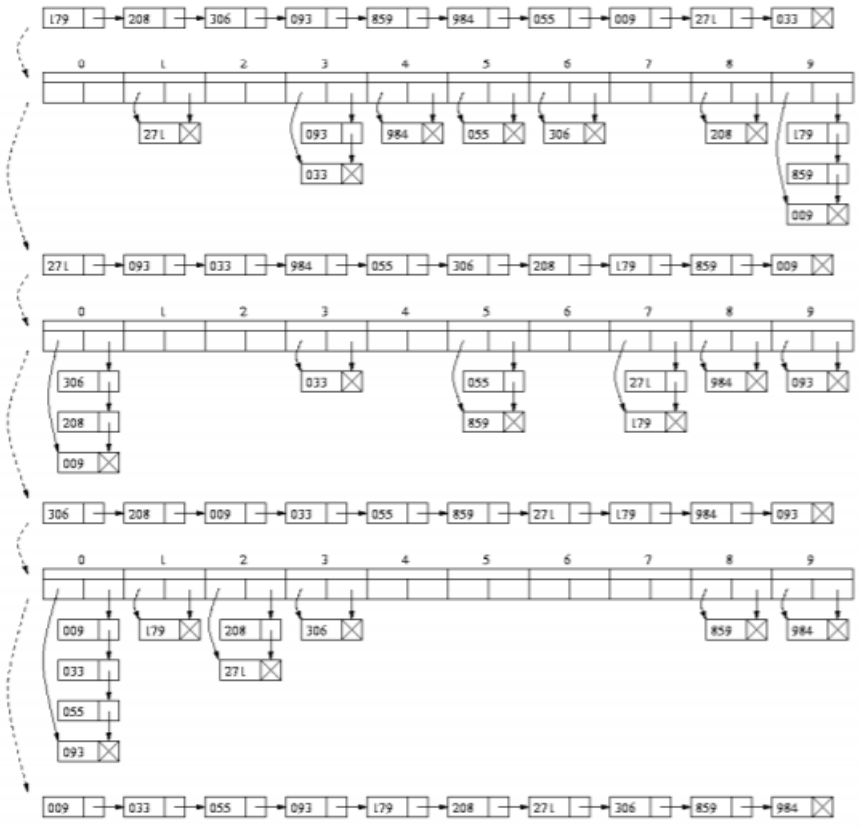
Questão 02:

3 Especificação do problema e explicação do método de solução

Abaixo estão as provas que utilizamos como base para nos auxiliar a provar no PVS.

Questão 01:

Um exemplo de como o RadixSort funciona está na imagem abaixo:



O radixsort ordenará com o auxílio de um algoritmo estável chamado *mergesort*. Para qualquer lista de p_1, p_2, ..., p_n, temos uma ordenação da lista L garantida pelo algoritmo estável.

Para d dígitos e $d = 1$, temos que a lista estará ordenada pois só há um dígito.

Se a lista está ordenada para $d = 1$ dígitos, deve ser verdade para até $k + 1$ dígitos pois:

Se ao chegar em k+1, se houver dois números diferentes a ordem será mantida baseado no último dígito k+1.

Se ao chegar em $k + 1$ os números forem iguais a ordem também será mantida pois utilizamos o algoritmo estável *mergesort* que preserva a ordem dos elementos.

Portanto pela hipótese de indução a ordem será mantida.

Questão 02

Para provar que a ordem é mantida, provaremos por indução no número de dígitos d.

Seja x um número de d dígito e x_l um número formado pelos últimos l dígotos de x para $x <= d$.

Para d = l, o RadixSort usa o *mergesort* como algoritmo estável para ordenar n números que vão de $0, 1, 2, ..., 9$. Portanto, temos três opções:

1. Se x_l for menor que y_l, x aparecerá antes de y.

2. Se x_l for igual a y_l as posições de x e y não serão modificadas pois usamos o *mergesort* que nos garante preservar a ordem.

3. Se x_i for igual a x_i, todos os dígitos que foram ordenados são os mesmos. Por indução, x e y permanecerão na mesma ordem e eles apareceram antes de i-ésima iteração, e a iteração é estável portanto eles devem permanecer assim após qualquer iteração adicional.

Questão 03

Utilizando o *mergesort* como algoritmo auxiliar, provaremos por indução que o radixsort mantém uma lista ordenada de tamanho X ou menor.

Uma prova simples do algoritmo *mergesort* será baseada em duas listas L.

Base: n = 1, lista de um elemento já está ordenada.

Hipóteste de indução: O mergesort funciona para $n = 1, 2, ...k$.

Passo indutivo: Funciona para $n = k + 1$ elementos.

O *mergesort* divide a lista em duas sublistas:

$L = [1, n/2]$ and $R = [n/2 + 1, n]$.

Temos que $(n/2)$ é menor que k. Por hipótese de indução os resultados de L e R estão ordenados. Além disso, a partir de nossa suposição, ao unir as listas será gerado uma matriz ordenada que contém todos os elementos porque tamanho de (L) + tamanho de (R) = n, o que significa que ordenou corretamente uma matriz de tamanho $n = k + 1$.

Com base nesta pequena prova do algoritmo *mergesort*, podemos por fim provar que o RadixSort manterá a ordem dos elementos.

Ao chamar a parte L e R que são sublistas produzidas pelo algoritmo *mergesort*, temos que cada metade é menor que o todo e portanto se a lista tiver um tamanho $X + 1$ cada metade não será maior que X. Isso é verdadeiro pois a prova do *mergesort* nos garante que para todas as listas de tamanho X ou menor estarão ordenadas.

Ao mesclar, construímos uma sequência repetida ordenada anexando a lista sempre o menor do primeiro elemento da metade esquerda e o primeiro elemento da metade direita. Estes elementos sempre serão os menores restantes da metade L ou R até que a sequência final esteja classificada.

Portanto, ao realizar o *mergesort* de uma lista X ou menor significa que está correto para todos os tamanhos $X + 1$ ou menor. Isto implica que também é correto para todas as listas de tamanho 1 ou menor. Por indução, é correto para todos os tamanhos.

4 Conclusões

Concluímos o trabalho de forma honesta com nossas limitações perante a ferramenta PVS porém registrado a nossa linha de raciocínio que aumentou o nosso conhecimento e nos desafiou um pouco mais.

O aprendizado deixado por este trabalho nos aproximou de uma ferramenta de provas formais, nos inseriu em textos e guias de alto nível.

Compreendemos a importância do uso de uma ferramenta para provas formais de algoritmos e o seu uso na lógica como um todo.

Bibliografia

M.Ayala-Rincon and F. L. C. de Moura. Applied Logic for Computer Scientists - Computational Deduction and Formal Proofs. Undergraduate Topics in Computer Science. Springer, 2017.

ALGORITMOS - Disponível em: ¡http://producao.virtual.ufpb.br/books/camyle/introducao-a-computacao-livro/livro/livro.chunked/ch05s01.html ¿. Acesso em: 13 jun. 2019.

PVS PROVER GUIDE - Disponível em: ¡http://pvs.csl.sri.com/doc/pvs-prover-guide.pdf ¿. Acesso em: 16 jun. 2019.

Data Structures, Spring 2004 L. Joskowicz 1 Data Structures – DAST Course 67109 and 67110 Spring 2004 Prof. Leo Joskowicz School of Engineering and.