Projeto Radix Sort (2019/1)

Eduardo Ferreira de Assis - 17/0102289
Guilherme Mendel de Almeida Nascimento - 17/0143970
Paulo Alvim Alvarenga - 17/0153657

18 de junho de 2019

Resumo

O presente relatório diz respeito a um projeto desenvolvido por alunos da matéria Lógica Computacional, ministrada pelo professor Flávio Moura, da Universidade de Brasília (UnB). O projeto consiste em verificar a formalização do algoritmo Radix Sort — para tal, o professor definiu 3 questões: teoremas que dizem respeito à sua estabilidade e consistência. A tarefa dos alunos foi de, por meio do assistente virtual de provas PVS, provar as questões e concluir a corretude do algoritmo.

1 Introdução

Algoritmos de ordenação de listas numéricas representam um tópico extremamente relevante para a comunidade científica da computação. Tal qual para qualquer outro problema de algoritmo, o objetivo dessa comunidade é encontrar a solução menos custosa e eficiente. Neste artigo, o algoritmo Radix Sort é apresentado e definido, 3 de seus teoremas base são abordados, e 2 destes têm sua prova explicitada, com o auxílio do assistente virtual de provas PVS.

Este artigo será divido em Introdução, Metodologia, Especificações dos Problemas e Soluções, Conclusão e Referências.

2 Metodologia e Base Teórica

O Radix Sort é um algoritmo de ordenação estável de itens com chave única (Wikipédia). Dada uma lista de strings composta de inteiros, por exemplo, sua aplicação resultará numa ordenação das strings conforme o valor de seus dígitos.

O Radix Sort pode ser de dois tipos:

- *LSD – Least significant digit radix sort* (Radix Sort por dígito menos significativo);
- *MSD – Most significant digit radix sort* (Radix Sort por dígito mais significativo)

O Radix Sort LSD começa a ordenação do dígito menos significativo (unidades, dezenas, centenas, ...), enquanto o MSD começa do dígito mais significativo (.., centenas, dezenas, unidades). O algoritmo em MSD é mais sensível, e é necessário mais cuidado em sua implementação, a fim de que não se desordenem dígitos já ordenados. A variação analisada neste projeto é a LSD.

Todas as provas foram realizadas por meio da utilização do PVS, que faz uso do sistema de sequentes de Gentzen. As teorias e os métodos aplicados foram extraídas das aulas do professor Flávio Moura, com base em seu trabalho Ayala-Rincón and De Moura, 2017, Applied Logic for Computer Scientists:Computational Deduction and Formal Proofs. Além disso, foram usados como base adicional para o desenvolvimento das provas literaturas ofertadas pelo professor:

- Baase, 2009, Computer algorithms: introduction to design and analysis. Pearson Education India;
- Cormen et al., 2001, Introduction to algorithms;
- Knuth, 1997, The art of computer programming: sorting and searching, volume 3.

3 Especificação dos problemas e soluções

Como citado anteriormente, para provar a corretude do algoritmo Radix Sort foram fornecidos 3 teoremas (*conjectures*) a serem provados. As duas primeiras comprovam uma porção da teoria base do algoritmo, enquanto que a terceira conclui que o algoritmo realmente realiza a ordenação de forma estável. Cada um desses teoremas foi separado em subseções para aprofundar sua explicação e a discussão da respectiva solução desenvolvida pelo grupo.

3.1 Questão 1

A questão 1 consiste em provar que dada uma lista *l*, se essa lista está ordenada até o (d-1)-ésimo dígito, então a aplicação do radix sort em um intervalo de *d* até *k* resultará em uma lista ordenada até o k-ésimo dígito. Em outras palavras, a ordenação por radix sort é construtiva e expande o conjunto ordenado existente.

```
radix_sort_d_sort   : CONJECTURE
 FORALL(l : list[nat], k : nat, d:nat | d <= k) :
   is_sorted_ud?(l,d) =>
      is_sorted_ud?(radixsort(l, k, d), k+1)
```

Figura 1: Questão 1, especificada em Lisp.

Na especificação acima, foi utilizada a função *is_sorted_ud?(l: list[nat], k: nat)* que confere se uma lista *l* está ordenada até o seu (k-1)-ésimo dígito (em representação decimal).

```
is_sorted_ud?(l : list[nat], d : nat) : bool = FORALL(i : below[length(l)], j : below[length(l)] | i <= j) :
    rem(10^d)(nth(l,i)) <= rem(10^d)(nth(l,j))
```

Figura 2: Declaração da função is_sorted_ud?, em Lisp

Observa-se também a utilização da função *radixsort(l: list[nat], k: nat, d: nat)*, que é o algoritmo previamente explicado Radix Sort, porém aplicado apenas dos dígitos *d* até *k* da lista *l*.

O primeiro passo no desenvolvimento de prova em PVS foi o comando *(measure-induct+ "k-d" ("k" "d"))*, sugerido pelo professor, que aplica indução forte sobre as variáveis explicitadas.

```
radix_sort_d_sort :

  |-------
[1]   FORALL (l: list[nat], k: nat, d: nat | d <= k):
        is_sorted_ud?(l, d) => is_sorted_ud?(radixsort(l, k, d), k + 1)
```

Figura 3: Questão 1 – Primeiro sequente.

Em seguida, *skeep* para remover quantificador do consequente e trazer a hipótese para o antecedente, utiliza as regras de Gentzen esquerda e direita para quantificadores. Depois, são aplicados alguns comandos de expansão, um de manipulação de condicionais (*expand*, *lift-if*), e então *prop*. Este comando aplica lógica de predicados e axiomas exaustivamente. A árvore de prova, então, se ramifica em 2 sequentes:

```
radix_sort_d_sort.1 :

{-1}  x!2 = x!1
[-2]  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)
[-3]  is_sorted_ud?(l, x!2)
  |-------
{1}   is_sorted_ud?(merge_sort(l, x!2), 1 + x!1)
```

Figura 4: Questão 1 – Primeiro sequente após o primeiro comando *prop*.

No consequente, o interessante é descobrir se a lista *l* ordenada por merge_sort até o digito x!2 está ordenada até 1 + x!1. Sabe-se que x!2 = x!1 e que a lista *l* esta ordenada até x!2 — logo, a aplicação do lema *"merge_sort_d_sorts"* resolve este sequente. O lema diz que uma lista *l*, ordenada até seu d-ésimo dígito, em que se aplica a função *merge_sort* (que ordena uma lista por um dígito *d*) sobre seu dígito *k* estará ordenada até o k+1-ésimo dígito.

Após a inserção do lema, algumas manipulações (*isnt?*, *prop*) bastam para fechar o sequente com o comando *assert* e concluir essa sub-árvore.

```
radix_sort_d_sort.2 :

[-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)
[-2]  is_sorted_ud?(l, x!2)
  |-------
{1}   x!2 = x!1
{2}   is_sorted_ud?(radixsort(merge_sort(l, x!2), x!1, 1 + x!2), 1 + x!1)
```

Figura 5: Questão 1 - Segundo sequente após o primeiro *prop*.

Nesse sequente, observa-se que a equação "x!2 = x!1" foi para o consequente. Sendo assim, concluir que ela é válida seria suficiente para provar esta sub-árvore, mas a princípio, isso seria impossível.

Através de instanciações de variáveis pelo comando *inst*, a equação "-1" no antecedente se aproxima da equação "2" no consequente.

```
radix_sort_d_sort.2 :

{-1}  x!1 - (1 + x!2) < x!1 - x!2 IMPLIES
       is_sorted_ud?(merge_sort(l, x!2), 1 + x!2) =>
        is_sorted_ud?(radixsort(merge_sort(l, x!2), x!1, 1 + x!2), 1 + x!1)
[-2]  is_sorted_ud?(l, x!2)
  |-------
[1]   x!2 = x!1
[2]   is_sorted_ud?(radixsort(merge_sort(l, x!2), x!1, 1 + x!2), 1 + x!1)
```

Figura 6: Questão 1 - Sequente após instanciações.

A seguir, foi aplicado *prop* que gerou duas ramificações:

```
radix_sort_d_sort.2.1 :

[-1]  is_sorted_ud?(l, x!2)
  |-------
{1}   is_sorted_ud?(merge_sort(l, x!2), 1 + x!2)
[2]   x!2 = x!1
[3]   is_sorted_ud?(radixsort(merge_sort(l, x!2), x!1, 1 + x!2), 1 + x!1)
```

Figura 7: Questão 1 – Primeiro sequente após o *prop*.

Novamente, basta aplicar o lema *"merge_sort_d_sorts"* para concluir essa sub-árvore. O ramo seguinte foi concluído com *isnt?* e *assert*.

```
radix_sort_d_sort.2.2 :

[-1]  is_sorted_ud?(l, x!2)
  |-------
{1}   x!1 - (1 + x!2) < x!1 - x!2
[2]   x!2 = x!1
[3]   is_sorted_ud?(radixsort(merge_sort(l, x!2), x!1, 1 + x!2), 1 + x!1)
```

Figura 8: Questão 1 – Segundo sequente após o *prop*.

Através do comando *assert* o PVS fecha a prova do sequente acima sozinho, pois o consequente ”1”é sempre verdadeiro, logo, trivialmente prova-se a validez do sequente.

A árvore de prova pode ser vista:

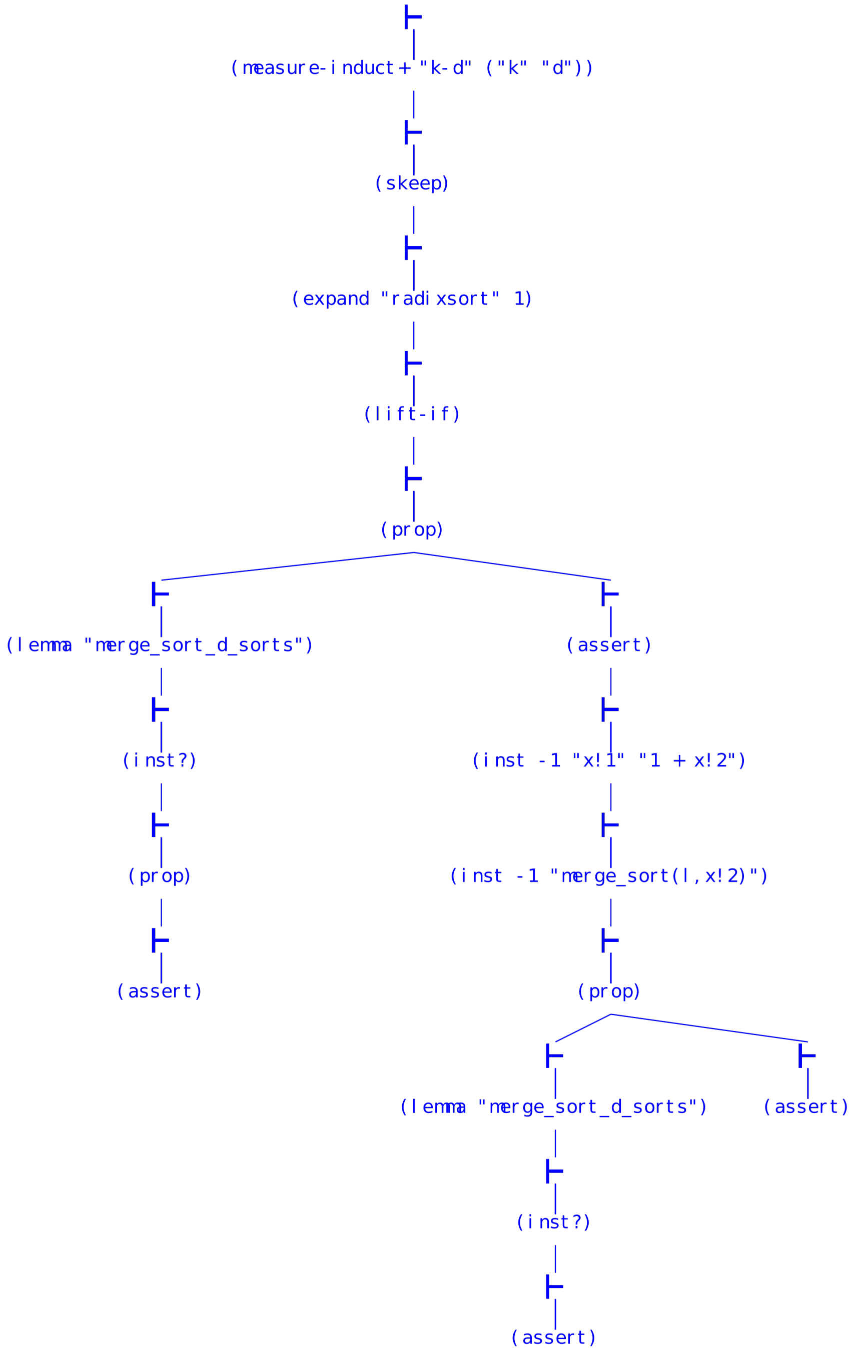

Figura 9: Árvore final.

3.2 Questão 2

A questão 2 consiste em provar que dado uma lista ”l”e dois naturais ”d”e ”k”, sendo ”d”menor ou igual a ”k”, a aplicação da função radixsort na lista ”l”entre o digito ”d”e o digito ”k”possui os mesmos elementos da própria lista. Em outras palavras, a permutação de radixsort(l, k, d) e ”l”é verdadeira.

```
radixsort_permutes : CONJECTURE
 FORALL(l : list[nat], k : nat, d : nat | d <=k):
 permutations(l, radixsort(l,k,d))
```

Figura 10: Questão 2, especificada em Lisp.

Para iniciar a prova, foi usado o comando ’(measure-induct+ ”k-d”(”kd”))’ tal como foi sugerido pelo professor. Isso significa que a prova é começada com uma indução forte em ”k-d”, gerando o seguinte sequente:

```
{-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
           permutations(l, radixsort(l, y_1, y_2))
  |-------
{1}   FORALL (l: list[nat]): permutations(l, radixsort(l, x!1, x!2))
```

Figura 11: Resultado da indução forte.

Então foi usado o comando *skeep* para simplificar o consequente, e foi expandida a função radixsort. Em seguida, utilizou-se *lift-if* seguido de *prop*, gerando duas sub-árvores.

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

Figura 12: Árvore à esquerda.

```
[-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
           permutations(l, radixsort(l, y_1, y_2))
  |-------
{1}   x!2 = x!1
{2}   permutations(l, radixsort(merge_sort(l, x!2), x!1, 1 + x!2))
```

Figura 13: Árvore à direita.

A árvore à esquerda foi provada através da introdução do lema *merge_sort_permutes*, que estabelece que data uma lista "l"qualquer e um natural qualquer "d", a permutação de "l"e o resultado da função merge_sort de "l"no digito "d"é verdadeira. Ou seja, merge_sort preserva os dados de l.

```
% Lemma states that merge_sort permutes (preserves) data.
  merge_sort_permutes : LEMMA
  FORALL(l : list[nat], d : nat):
  permutations(l,merge_sort(l,d))
```

Figura 14: Merge_sort_permutes em lisp.

Instanciando este lema com "l"e "x!2", foi possível perceber que o antecedente e consequente seriam iguais, logo, esta árvore pode ser provada.

A prova da árvore à direita, por sua vez, também utilizou este lema, e sua instanciação da mesma forma.

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

Figura 15: Primeira instanciação da arvore à direita.

A segunda instanciação foi do antecedente gerado na indução forte. A variável "y_1"foi instanciada como "x!1"e "y_2"como "1 + x!2", "l"foi instanciado como "merge_sort(l,x!2)"formando o sequente abaixo:

```
[-1]  permutations(l, merge_sort(l, x!2))
{-2}  x!1 - (1 + x!2) < x!1 - x!2 IMPLIES
       permutations(merge_sort(l, x!2),
                    radixsort(merge_sort(l, x!2), x!1, 1 + x!2))
  |-------
[1]   x!2 = x!1
[2]   permutations(l, radixsort(merge_sort(l, x!2), x!1, 1 + x!2))
```

Figura 16: Sequente após instanciações da arvore à direita.

É realizado um *assert* pois a primeira condição de -2 é sempre verdade. É introduzido então o lema ”permutations_is_transitive”que estabelece que caso tenha-se duas permutações de listas verdadeiras, tal como l,l1 e l1,l2 é possível concluir que a permutação de l,l2 é também verdadeira (transitividade da permutação).

```
%permutations is transitive
permutations_is_transitive : LEMMA
permutations(l,l1) AND permutations (l1,l2) IMPLIES permutations (l,l2)
```

Figura 17: Lema de permutations_is_transitive.

Este lema foi instanciado com ”l”, ”merge_sort(l,x!2)”e ”radixsort(merge_sort(l,x!2), x!1, 1+x!2)”para ”l”, ”l1”e ”l2”, respectivamente pois já se possuía como hipótese a permutação de ”l,l1”e ”l1,l2”, podendo concluir a permutação de ”l,l2”que é justamente o consequente que se queria provar. Após um *assert*, este sequente é provado.

4 Conclusão

A importância de uma formalização se torna ainda mais evidente uma vez que o desenvolvedor contempla a quantidade de possibilidades de falha presentes num algoritmo. Após a realização deste projeto, os alunos concluíram que certamente só se deve confiar num algoritmo cuja corretude seja formalmente comprovada.
Infelizmente, nem sempre os desenvolvedores possuem a sensibilidade necessária para levar em consideração as consequências e riscos que um algoritmo não formalizado pode apresentar — desde erros na segurança de um sistema bancário até a explosão preventiva de um foguete não funcional.

Referências

Ayala-Rincón, M. and De Moura, F. L. (2017). *Applied Logic for Computer Scientists: Computational Deduction and Formal Proofs*. Springer.

Baase, S. (2009). *Computer algorithms: introduction to design and analysis*. Pearson Education India.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C., et al. (2001). Introduction to algorithms.

Knuth, D. E. (1997). *The art of computer programming: sorting and searching*, volume 3. Pearson Education.

Wikipédia. Radix sort – wikipédia, a enciclopédia livre. https://pt.wikipedia.org/wiki/Radix$_sort.(Accessedon06/15/2019)$.