

Projeto Radix Sort (2019/1)

Allann Gois Hoffmann 180029789
Artur de Meira Rodrigues 180013688

17 de junho de 2019

1 Introdução

O projeto tem como finalidade mostrar a funcionalidade do radix sort, por meio do assistente de demonstração PVS e provar que o mesmo funciona para qualquer lista fornecida. O radix sort é um algoritmo de ordenação estável, na qual retorna todos os dígitos fornecidos de forma ordenada. O algoritmo ordena do dígito menos significativo para o dígito mais significativo, que seria do dígito das unidades até o maior dígito.

2 Contextualização

No trabalho existem 3 questões que pedem as seguintes provas:

- A primeira questão pede para provarmos que uma lista está ordenada até o d -ésimo dígito se aplicando ao radix sort entre os dígitos D e K , assim ela ficará ordenada até o dígito $k + 1$.

- Seguindo com a segunda questão precisamos demonstrar que a função radix sort preserva os elementos das listas.

- E a terceira e última questão é demonstrar que o radix sort ordena listas perfeitamente.

3 Questões

3.1 Questão 1

- Expand “radixsort” 1: Ele expandiu o radixsort da linha 1 de acordo com a definição dele dada pelo projeto.

```

Sequent 13 (radix_sort_d_sort)

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

```

Figura 1: Radix sort

-Lift-if: De acordo com a própria tradução ele separa o IF do consequente. Vimos a possibilidade de separar o IF no consequente e conseguir abrir os ramos na árvore.

```

Sequent 2 (radixsort_permutes)

radixsort_permutes :
[-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} IF x!2 = x!1 THEN permutations(l, merge_sort(l, x!2))
  ELSE permutations(l, radixsort(merge_sort(l, x!2), x!1, 1 + x!2))
  ENDIF

```

Figura 2: Lift-If

-Assert: Simplifica a função e finaliza a prova caso for possível. Sempre que possível utilizávamos ele para simplificar a prova e facilitar a visualização de uma resolução.

-prop: Ele abriu nossa prova em 2 ramos. O usamos pois existia um IF e ELSE, assim abriríamos os 2 ramos de acordo com o IF e ELSE.

-Aplicamos “hide -1” para “esconder” a linha -1. Ao nosso ver ele não era útil na resolução, então decidimos “esconde-lo”.

-Analisando os lemas disponíveis visualizamos que o lemma “merge sort d sorts” tem a estrutura perfeita para a situação, só seria necessário uma instanciação para conseguirmos provar este ramo.

```

Sequent 12 (radix_sort_d_sort.1)

radix_sort_d_sort.1 :
{-1} FORALL (l: list[nat], d: nat):
  is_sorted_ud?(l, d) => is_sorted_ud?(merge_sort(l, d), d + 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), 1 + y_1)
[-3] is_sorted_ud?(l, x!2)
|-----
[1] is_sorted_ud?(merge_sort(l, x!2), 1 + x!1)

```

Figura 3: Lemma “merge sort d sorts”

-Segundo ramo: -Começamos com um “assert” para simplificar. -Instanciamos automaticamente (como na última instanciação o “inst?” bastou).

```

Sequent 15 (radix_sort_d_sort.2)

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 4: Inst?

-Vimos que podíamos dar “hide 1” pois a linha 1 não era essencial para provar o ramo.

-Novamente o “assert” para simplificar a questão.

-Outra vez o “hide 1” para “esconder” o que não era essencial.

-Como no primeiro ramo o lema “merge sort d sorts” foi o essencial para conseguirmos a estrutura necessária e finalizar a prova.

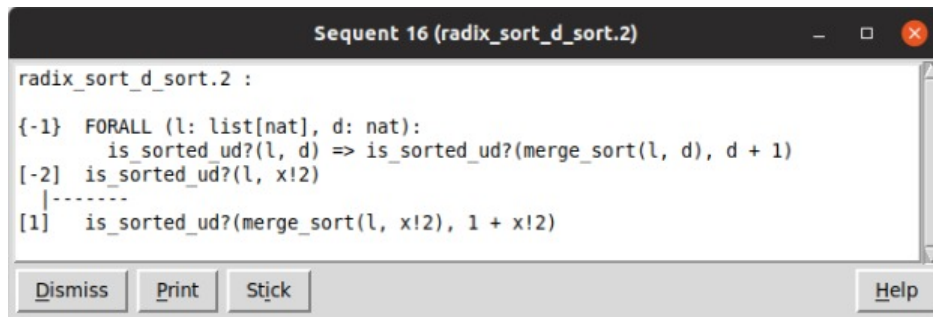


Figura 5: Lemma "merge sort d sorts"

-Já quase no final foi somente necessário o "inst?" para instanciar o lema com as informações da questão e enfim conseguir finalizá-la.

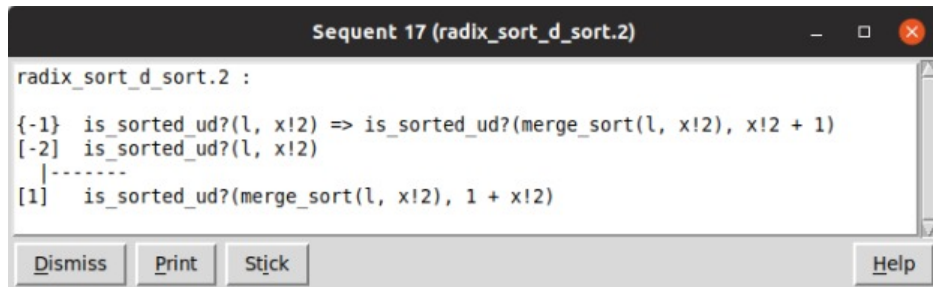


Figura 6: Inst?

-E por fim um assert para fechar a questão 1.

3.2 Questão 2

-Como na questão 1, foi aplicado os mesmos comandos até o passo "prop-Em seguida, usamos o lema "merge sort permutes", na qual ele tinha a mesma estrutura do consequente

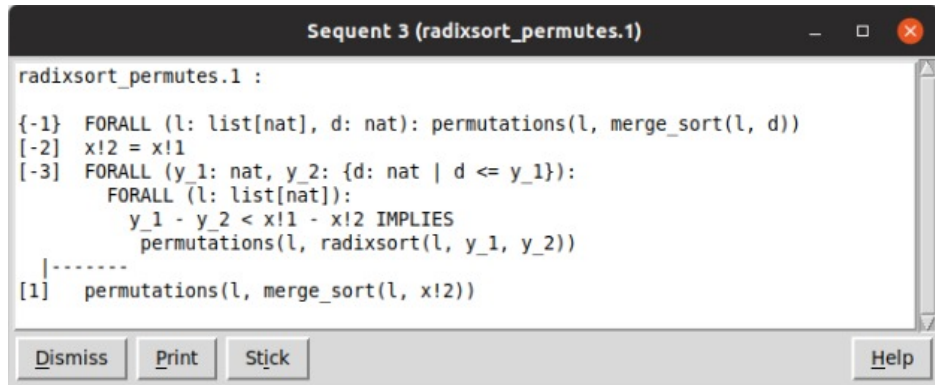


Figura 7: Lemma "merge sort permutes"

-E para fechar o ramo, foi necessária uma instanciação automática. -No segundo ramo: -Foi instanciada automaticamente a função "permutations" -l

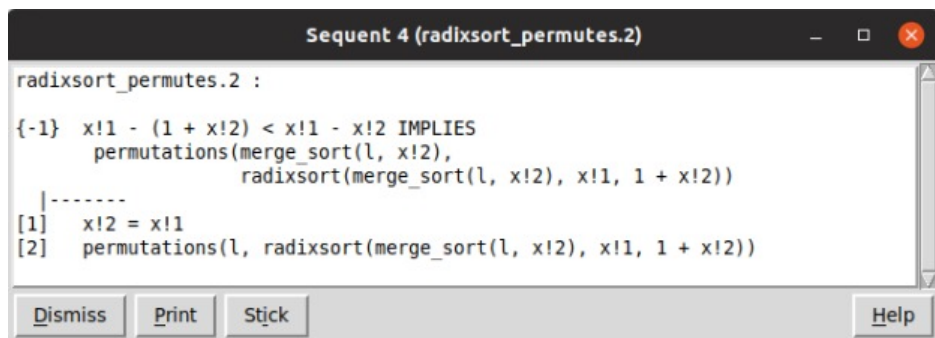


Figura 8: Inst?"

-Em seguida o lemma "Merge sorts permutes" novamente

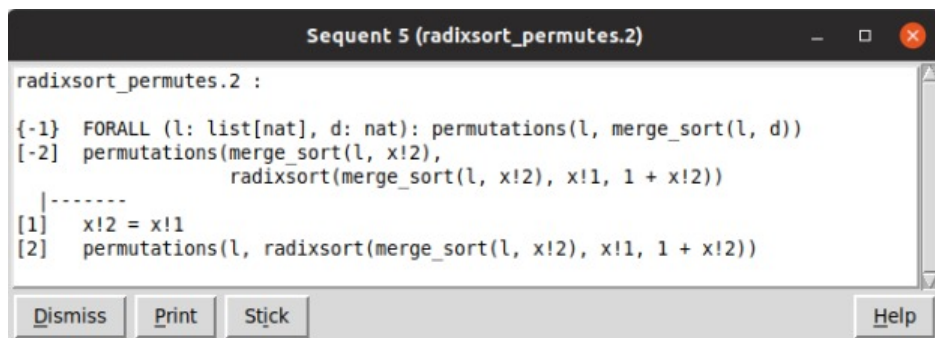
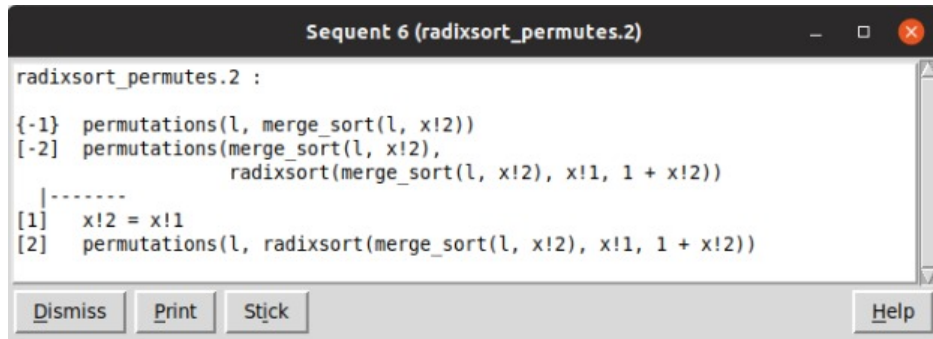


Figura 9: Lemma "merge sort permutes"

-Logo após fizemos a instanciação com o lemma



```
Sequent 6 (radixsort_permutes.2)

radixsort_permutes.2 :

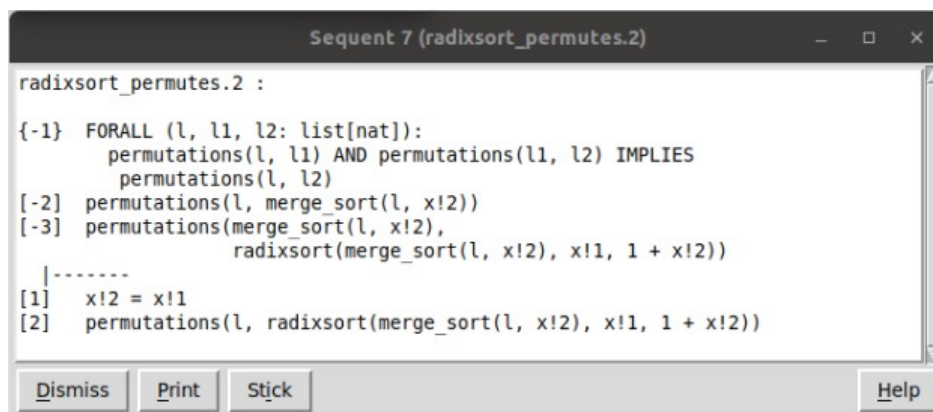
{-1} permutations(l, merge_sort(l, x!2))
[-2] permutations(merge_sort(l, x!2),
                radixsort(merge_sort(l, x!2), x!1, 1 + x!2))

|-----
[1]  x!2 = x!1
[2]  permutations(l, radixsort(merge_sort(l, x!2), x!1, 1 + x!2))

Dismiss Print Stick Help
```

Figura 10: Inst?

- Aplicamos o lemma encontrado na pasta Sorting.pvs chamado “permutations is transitive”, pois se “A” permuta com “B” , “A” permuta com “C”



```
Sequent 7 (radixsort_permutes.2)

radixsort_permutes.2 :

{-1} FORALL (l, l1, l2: list[nat]):
    permutations(l, l1) AND permutations(l1, l2) IMPLIES
    permutations(l, l2)
[-2] permutations(l, merge_sort(l, x!2))
[-3] 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))

Dismiss Print Stick Help
```

Figura 11: Lemma”permutations is transitive”

-Instanciamos o lemma trago com, “l” sendo “l”, “l1” com “merge sort”, ”l2” com “radix sort”.

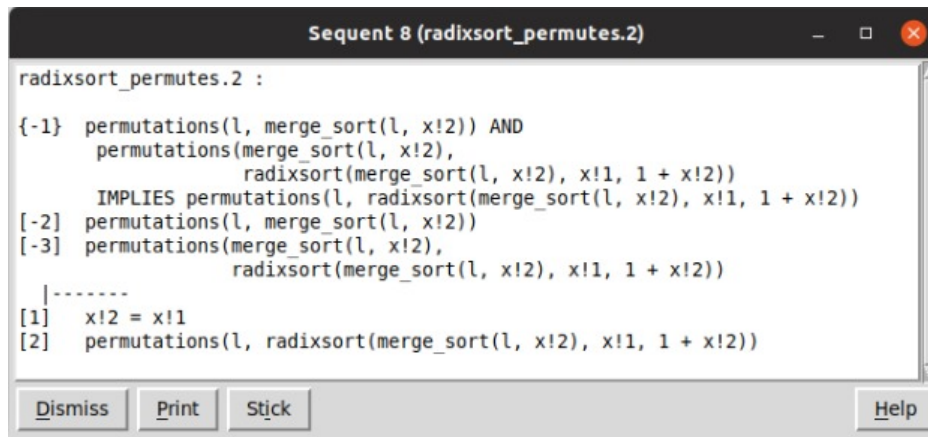


Figura 12: Inst

- E por fim utilizado um assert para fechar o ramo 2.

4 Problemas e Métodos

Os principais problemas encontrados na resolução do projeto foi achar formas de solucionar as questões de modo geral, sendo que encontrar os lemmas certos e as instanciações no seu devido lugar foi a principal dificuldade. Para solucionar, começamos a fazer a mão utilizando técnicas dedutivas da logica de predicados. O avanço no papel foi significativo porem, ao passar o resultado para o PVS, encontramos problemas relativo a linguagem do assistente de provas.

5 Conclusão

Podemos concluir que o Radix Sort funciona para qualquer lista que passar pelo algoritmo, e por meio do assistente de provas PVS foi possivel provar de forma eficaz e facil, mesmo tendo dificuldades ao utiliza-lo, e conseguimos adquirir conhecimentos sobre o algoritmo e o programa PVS.