Formalização de propriedades do algoritmo Radix Sort

Grupo 9
Leonardo Rodrigues de Souza - 17/0060543
Lucas Vinicius Magalhães Pinheiro - 17/0061001
Luthiery Costa Cavalcante - 17/0040631
Thiago Veras Machado - 16/0146682

[1]Dep. Ciência da Computação – Universidade de Brasília (UnB)
CiC 117366 - Lógica Computacional 1 - Turma A

Abstract. *This paper talks about the sorting algorithm "Radix Sort", which consists in sorting the numbers taking in count only one digit of each for each turn, going from the least significant one to the most significant one (between all the numbers). In each step is used a secondary sorting algorithm and, in this paper, we chose to analyse Merge Sort. Specifically, the goal of the work is to prove that Merge Sort is fit to be used as a support for Radix Sort. For this work, we used the PVS proof assistant.*

Resumo. *Este trabalho se trata do algoritmo de ordenação Radix Sort, que consiste em ordenação considerando apenas um dígito de cada elemento por vez, partindo do menos significativo até o mais significativo. Em cada passo é usado um algoritmo secundário de ordenação e, neste trabalho, escolhemos analisar o Merge Sort. Mais especificamente, o objetivo do projeto é provar que o Merge Sort está apto a ser utilizado como auxiliar ao Radix Sort. Nosso ambiente de trabalho é o assistente de provas PVS.*

Brasília
22 de novembro de 2018

1. Introdução

A ideia do *Radix Sort* é ordenar um grupo de números partindo do primeiro (menos significativo) dígito de cada um, depois repetir o procedimento para as dezenas, centenas e assim em diante até o dígito mais significativo. Os números não precisam ter a mesma quantidade de dígitos. Em cada passo é usado um algoritmo secundário de ordenação que ordena considerando apenas o d-ésimo dígito de cada número, onde d vai de 0 até a maior quantidade de dígitos encontrada na lista. Há uma restrição para este algoritmo auxiliar: ele deve ser *estável*. Dizemos que um algoritmo de ordenação é estável se a posição relativa de dois elementos iguais permanece inalterada durante o processo.

O *Radix Sort* pode ser estendido a outras estruturas além de números. Por exemplo, suponha que queremos ordenar (classificar) dados dos alunos de LC1 em uma planilha. Cada aluno tem nome, matrícula e nota, entre outros. É possível fazer uma ordenação dessa lista pelo nome, depois pela matrícula e por nota, onde cada ordenação intermediária é feita segundo algum método específico.

Este projeto aborda a formalizações de propriedades do *Merge Sort* que o qualificam como algoritmo estável, apto a ser usado como algoritmo secundário pelo *Radix Sort*. A formalização das propriedades é feita usando o assistente de provas PVS encorpado com a biblioteca `nasalib`. [1]

O PVS é um assistente de provas que consiste em uma linguagem de especificação, que é usada para especificar teorias, conjecturas, lemas, funções, procedimentos, predicados e expressões lógicas; e uma linguagem de prova, que é responsável por aplicar regras para provar as propriedades. Tanto os comandos de prova quanto a própria estrutura das provas estão correlacionadas com o Cálculo de Sequentes e, por isso, várias noções sobre esse método dedutivo visto em aula e nas notas de aula [2] serão úteis no trabalho e vice-versa. O PVS é uma ferramenta de prova muito poderosa, porém esse projeto foi elaborado de forma que necessitemos apenas da Lógica de Primeira Ordem (LPO).

Na próxima seção, listaremos algumas funções e proposições importantes ao longo do trabalho. Na seção 3 constam o raciocínio e os passos para concluir as provas duas primeiras questões. Por último, na seção 4, há algumas observações sobre o resultado atingido e no que isso ajuda a deduzir sobre o algoritmo *Radix Sort*.

2. Definições

No arquivo principal do projeto, *radix_sort.pvs*, estão listadas todas as conjeturas a serem provadas na seção 3, além de várias definições de funções e lemas que possam ser úteis para prová-las. Para melhor contextualizar o leitor, os mais relevantes ao nosso desenvolvimento serão aqui explicados e, se necessário, definidos abaixo.

- `n_digits()`

```
n_digits(n:nat) : RECURSIVE posnat =
  IF n < 10 THEN 1
  ELSE 1 + n_digits(ndiv(n, 10))
  ENDIF
  MEASURE n
```

  Como sugere o nome, a função conta recursivamente o número de dígitos de um número natural n. Nessa definição aparece $ndiv(x, n)$, cujo resultado é a divisão inteira de x por n, sendo x e n naturais e $n > 0$.

- `merge_sort()`

```
merge_sort(l : list[nat], d :nat) : RECURSIVE list[nat] =
  IF length(l) <= 1 THEN l
  ELSE merge(merge_sort(prefix(l,floor(length(l)/2)), d),
             merge_sort(suffix(l,floor(length(l)/2)), d), d)
```

```
  ENDIF
  MEASURE length(l)
```

Função de grande importância para o entendimento do projeto, ela divide uma lista l em duas metades - de forma a não se perder elementos mesmo que haja uma quantidade ímpar deles - e chama *merge()* para ordenar essas metades baseado no d-ésimo dígito dos números, retornando essa nova lista ordenada. Ela repete esse processo até que l original esteja toda fragmentada em listas unitárias.

- `merge()`

```
merge(l1, l2 : list[nat], d : nat) : RECURSIVE list[nat] =
IF null?(l1) OR null?(l2) THEN append(l1, l2)
ELSIF d_nth(car(l1),d) <= d_nth(car(l2),d) THEN
  cons(car(l1), merge(cdr(l1),l2, d))
ELSE cons(car(l2), merge(l1, cdr(l2), d))
ENDIF
MEASURE length(l1) + length(l2)
```

A função recebe as duas listas l_1 e l_2 de mergesort() e compara seus topos(*car()*). Se o topo (ou melhor, o d-ésimo dígito do topo) de l_1 é menor ou igual ao de l_2, a união dessas duas listas terá o topo de l_1 na frente. Então, ela continua a ordenação chamando-se recursivamente com o que restou das duas listas. Análogo ao que ocorre se o topo de l_2 for menor que o de l_1. A imagem abaixo ilustra um funcionamento de *merge sort* + *merge*, que ordena uma lista comparando os números por si, isto é, não há a ideia de d-ésimo digito que empregamos nesse projeto.

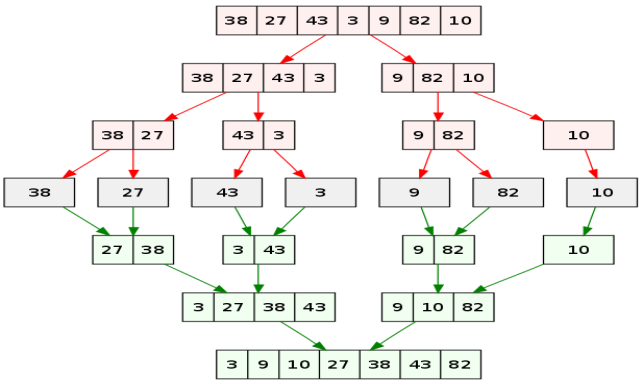

Figure 1. Exemplo de Merge Sort. *retirado de [3]*

- `permutations()` Duas listas (ou sequências) são permutação uma da outra se os elementos delas são os mesmos. Isso é feito contando o número de vezes que cada elemento aparece na estrutura da lista (ou sequência). Em outras palavras, duas listas l_1 e l_2 são permutação uma da outra se

$$\forall x \text{ ocorrências de } x \text{ em } l_1 = \text{ocorrências de } x \text{ em } l_2.$$

- `occurrences_of_app` Esse lema afirma que

$$\forall l_1, l_2, x(\text{ ocor. de } x \text{ em } append(l_1, l_2) = \text{ ocor. de } x \text{ em } l_1 + \text{ ocor. de } x \text{ em } l_2)$$

e será útil na resolução da questão 2.

- `is_sorted_ud?` A função retorna verdadeiro se a lista está ordenada com respeito ao d-ésimo digito.

3. Soluções

Solucionamos as duas primeiras questões. A prova da segunda é ligeiramente mais extensa. Lá aparece o único lema usado em nossa prova, *occurrences_of_app*.

3.1. Questão 1: `d_digits_gt`

Na primeira questão, nós temos que provar que para todo x, $10^{(\text{número de dígitos de x})} > x$, uma ideia intuitivamente correta que parece sempre ser válida.

```
FORALL(n : nat):
  10^(n_digits(n)) > n
```

Mas é preciso formalizar a prova. E segundo o sugerido, o primeiro passo é aplicar indução forte em n. Com isso, tentaremos provar que essa conjetura é válida para qualquer $y < x$. Como no antecedente e no consequente as variáveis trabalhadas são diferentes, por hora nós expandimos a definição de *n_digits()* no consequente, para podermos em seguida trabalhar com cada um dos casos, onde $x < 10$ e onde $x >= 10$.

```
[-1]  FORALL (y: nat): y < x!1 IMPLIES 10 ^ (n_digits(y)) > y
  |-------
{1}   IF x!1 < 10 THEN 10 ^ 1 > x!1
      ELSE 10 ^ (1 + n_digits(ndiv(x!1, 10))) > x!1
      ENDIF
```

O primeiro caso é trivial: queremos mostrar que se $x < 10$ então $10 > x$.

No segundo caso, nós voltamos no problema original com a diferença de que agora temos $ndiv(x!1, 10)$, então para facilitar o restante da prova, renomearemos o y no antecedente por $ndiv(x!1, 10)$, uma vez que nesse caso $x >= 10$, e portanto $ndiv(x, 10)$ sempre será menor que x (requisito de y). `(inst -1 "ndiv(x!1, 10)")`

Mais uma vez chegamos numa situação em que é necessário bifurcar a árvore para provar os possíveis casos, então aplicamos `(split -1)` para fazermos a divisão das duas novas subprovas.

Nesse primeiro caso, tínhamos no antecedente a informação $10^{\text{n\_digits(ndiv(x, 10))}} > ndiv(x, 10)$, e no consequente tínhamos $10^{\text{1 + n\_digits(ndiv(x, 10))}} > x$. Podemos reescrever isso como $10 \times 10^{\text{n\_digits(ndiv(x, 10))}} > x$, e também podemos reescrever a fórmula do antecedente como $10 \times 10^{\text{n\_digits(ndiv(x, 10))}} > 10 \times ndiv(x, 10)$. Se formos analisar essa ultima fórmula, é a mesma coisa de escrevermos $10 \times 10^{\text{n\_digits(ndiv(x, 10))}} > x$, então nosso primeiro palpite foi de que como havíamos chegado em um ponto onde havia uma igualdade lógica entre uma fórmula no antecedente e no consequente, podíamos chamar `(grind)` e fechar essa prova, contudo isso não foi possível. Após discussões com o professor a respeito dessa etapa, nós aproveitamos o subproduto do `(grind)` nesse mesmo nó, onde ele substitui o símbolo de potenciação (^) por *expt()*, obviamente mantendo a propriedade da operação, e daqui seguimos esse ramo da prova.

Por questões de melhor visualização do problema, em seguida nós renomeamos $expt(10, n\_digits(ndiv(x!1, 10)) - 1)$, que estava presente em ambos antecedente e consequente por K. Como ainda não conseguíamos achar uma relação direta a ser estabelecida, jogamos `typepred` para buscarmos na definição de $ndiv()$ informações que, mesmo que redundantes e óbvias, poderiam ser úteis para a conclusão da prova.

Nesse momento aplicamos `assert` para fechar esse ramo. Durante a apresentação do trabalho, foi sugerido retirar uma das fórmulas das três premissas e verificar se ainda era possível se chegar à conclusão. As fórmulas que restaram depois de retirarmos uma delas não foram o suficiente para se fechar a prova. Fica claro que as três eram sim necessárias para a conclusão, mas não pudemos enxergar direito o porquê.

Do outro lado, ficamos sem nenhuma hipótese e ainda tínhamos que provar $10^{1 + \text{n\_digits(ndiv(x, 10))}} > x$. Fizemos novamente um `typepred` para enriquecer o contexto. Lá temos agora $x = 10 \times ndiv(x, 10) + resto(x, 10)$ e $ndiv(x, 10) <= x$. No consequente tínhamos, além do problema principal, $ndiv(x, 10) < x$. Essas duas últimas são equivalentes sempre que $x \neq 0$, então assumimos que o `(grind)` seria inteligente para separar os dois casos. Em cada um, uma das fórmulas do consequente seria provada e, portanto, esta sub-árvore e a questão estão resolvidas.

3.2. Questão 2: `merge_permutes`

```
FORALL(l1, l2 : list[nat], d : nat):
permutations(append(l1,l2),merge(l1,l2,d))
```

A questão 2 trata de uma propriedade da função *merge()*. Temos a tarefa de provar que o *merge()* de duas listas l_1 e l_2 é uma permutação da concatenação (*append()*) de l_1 e l_2, isto é, preserva seus elementos - não há números apagados ou produzidos durante o *merge()*. Tanto sua definição quanto a de *permutations()* são encontradas na seção 2.

Aplicamos indução forte na soma dos tamanhos de l_1 e l_2, pois há a necessidade de aplicar a indução em uma fórmula que diga respeito a ambos l_1 e l_2, além de que essa é exatamente a medida da *merge()*, uma função recursiva. Nosso problema então se transforma em provar, a partir da hipótese

$$\forall y_1, y_2, d\ (len(y_1) + len(y_2) < len(l_1) + len(l_2) \rightarrow permutations(append(y_1, y_2), merge(y_1, y_2, d))$$

que $merge(l_1, l_2, d)$ é permut. de $append(l_1, l_2)$. Usaremos x_1 e x_2 para nos referir a l_1 e l_2, respectivamente, para manter conformidade com a prova do PVS.

Para desenvolver a prova, seguindo a definição de *merge()*, temos que provar cada um dos seguintes casos:

- Caso x_1 ou x_2 sejam vazias ficamos com a tarefa de provar que $append(x_1, x_2)$ é permutação de $append(x_1, x_2)$, trivialmente verificável pois uma lista sempre é permutação de si mesma.

- Caso nenhuma das duas seja vazia, compara-se o topo (*car()*) das duas listas. O elemento que for menor ou igual estará mais à "frente" no resultado final. Este é o caso em que $car(l_1) <= car(l_2)$. Embora no arquivo enviado a prova desse caso é um simples *(grind)*, que usamos como um atalho no momento, podemos fazer uma expansão para que a prova seja mais detalhada.
  Esta etapa consiste em explicar se a construção da lista de *cabeça* de x1 com o *merge* da *calda* de x1 com a lista x2 é *permutação* de *append* de x1 com x2. Para isso, podemos expandir a definição de *permutations* e depois expandir a definição de *occurrences*, feito estes passos, precisamos provar os 3 sub casos da função *occurrences*. O primeiro sub caso precisamos provar 3 fórmulas, a segunda temos *null?(x1)* e a terceira *null?(x2)*, como antecedente temos que *null?(append(x1, x2))* então se temos que a junção de 2 listas é nula, logo cada 1 individualmente é nula, por ser trivial, um *grind* resolve.

  A segunda sub-árvore precisamos provar que 1 + *ocorrências* de *x* da *calda* do *append* de x1 com x2 é igual a 2 sub casos, caso a *cabeça* desse *append* for igual à *x*, entao a reposta é 1 + *ocorrências* de x no merge do que tinhamos antes sem a cabeça de *x1*, se não, o mesmo que antes sem somar 1, pois como a *cabeça* não é x, logo essa *ocorrência* não será contada. Para resoulver este ramo, basta olhar para o antecedente e ver que a *cabeça* do *append* de x1 e x2 é x podemos concluir que a *cabeça* de x1 é x, pois pela definição de *append* sabemos que ele apenas concatena 2 listas, sendo assim no consequente entraremos na primeira condição que fala caso a cabeça de x1 seja x, e com isso basta instanciar a fórmula -3 que consistem em se temos 2 listas que sua soma é menor que a lista x1 e x2, entao a *ocorrência* de x no

append dessas 2 listas é igual as *ocorrências* de x no *merge* dessas 2 listas. Ao instanciar y1 com a *calda* de x1 e y2 como o próprio x2 chegamos em uma parte trivial no antecedente e consequente, necessitando de apenas um *grind*.

A terceiro e ultima sub-árvore se assemelha à subàrvore 2 então a explicação já foi realizada.

- No último caso, a gente tem no consequente a função *merge()* que recebe como argumentos x_1 e $cdr(x_2)$, então instanciamos essas duas listas no lugar de y_1 e y_2 no antecedente, pois provavelmente isso iria facilitar no futuro, já que $length(x_1) + length(cdr(x_2))$ obviamente será menor que $length(x_1) + length(x_2)$.
  No próximo nó não havia nada que pudesse ser obviamente reconhecido como o próximo passo, então nós expandimos a definição de *permutations()*. Com isso nós temos uma implicação à esquerda, então podemos usar o comando `prop` para separar nas duas subárvores. Mas primeiramente usamos `skeep` para retirar a universal à direita, para simplificar antes de passarmos o `prop`.
  Na primeira árvore, instanciamos no antecedente o x, já que ele também estava sendo usado no consequente. Após isso novamente vimos em uma parte onde não era muito claro o próximo passo, então seguindo o conselho dado pela própria questão de que alguns lemas sobre *length()* e *append()* seriam úteis, e o lema *occurrences_of_app* trata desses. Em seguida, precisávamos instanciar no antecedente algo que estivesse no consequente para podermos nos aproximar da conclusão, então novamente instanciamos x_1 e $cdr(x_2)$.
  No outro ramo, novamente nos encontramos sem perceber de imediato a próxima ação, e após tentar com vários lemas que tratassem sobre *length()*, tentamos aplicar justamente a definição dela nesse passo. Então usamos `lift-if` e `prop` para novamente dividir essa prova em duas sub-provas. Ambas são resolvidas com `grind`.

3.3. Questão 3: `merge_sort_d_sorts`

```
FORALL(l : list[nat], d : nat) :
is_sorted_ud?(l,d)  =>
is_sorted_ud?(merge_sort(l,d),d+1)
```

A questão 3 foi a questão que mais tivemos dificuldades, inclusive não conseguimos terminar sua prova. Esta questão se trata em provar a estabilidade da função merge_sort, sendo assim, precisamos provar que, se temos uma lista ordenada até o dígito d (sem incluí-lo) então se ordenarmos utilizando o merge sorte com base neste dígito d então a nova lista também estará ordenada para o dígito d + 1 (sem incluí-lo).

Utilizamos para a prova a indução forte na lista l, com isso temos no consequente um quantificador universal, realizamos a eliminação do universal a direita, então chegamos em um ponto onde tínhamos 2 fórmulas no antecedente:

- *Para toda* lista y e *para todo* número natural d, se o *tamanho* da lista y for menor que o *tamanho* da lista x1 implica que o que queríamos provar, só que neste caso para um subfórmula menor que é uma extritamente menor que a lista x1
- A lista x1 está *ordenada* até o dígito d (não incluso)

Com isso, assumimos que seria mais interessante expandir o *merge_sort* pois nosso espaço amostral não estava rico o suficiente. Ao expandir, dividimos a prova em 2 sub-casos, no qual tinhamos:

- *Para todo* natural d, se está *ordenado* até o dígito d (sem incluí-lo) então o *merge_sort* com base no dígito d também irá estar *ordenado* até o dígito d + 1 (sem incluí-lo).

Para provar esse ramo, bastou realizar a eliminação do quantificador universal a direta e com isso precisamos provar se a lista x1 está *ordenada* até o dígito 1 + d (sem incluí-lo), como no antecedente temos qque *tamanho* da lista x1 é *¡= 1*, por definição de *ordenação*, se a lista possui tamanho menor ou igual a 1, então essa lista já está *ordenada*, por ser trivial, apenas um *grind* foi necessário.

- Provar que está *ordenado* até o digito d + 1 o resultado do *merge* do *merge_sort* das 2 metades da lista x1, com isso achamos necessário o uso do lema *merge_preserves_sort*

```
merge_preserves_sort : LEMMA
  FORALL(l1,l2 : list[nat], d : nat):
  (is_sorted_ud?(l1,d+1) AND is_sorted_ud?(l2,d+1) AND
   FORALL(i:below[length(l1)], j:below[length(l2)]) :
    rem(10^d)(nth(l1,i))  <= rem(10^d)(nth(l2,j)) )  =>
   is_sorted_ud?(merge(l1,l2,d),d+1)
```

Como precisávamos mostrar que o resultado do *merge* de 2 listas resultantes de uma ordenação está ordenada, basta mostrar que o próprio algoritmo mantém a ordenação. Infelizmente a expansão das fórmulas não foram feitas na maneira correta, ficando preso em etapas redundantes fazendo que a prova não fosse concluída.

4. Conclusões

Com as provas prontas das duas primeiras questões e assumindo que a 3ª propriedade também é verdadeira, além de todos os lemas presentes, acreditamos haver todo o aparato teórico necessário para se provar a correção do *Radix Sort* usando *Merge Sort*, prova essa que aparentemente já existe, mas foi omitida do arquivo do projeto.

Entre os desafios enfrentados destacamos a dificuldade para entender algumas funções complicadas, como elaborar e chegar ao final da prova, incluindo o momento certo para o uso de `typepred` e lemas, e evitar atalhos preguiçosos no percurso, como o comando `(grind)`. Obstáculos esses que não conseguimos superar na questão 3.

References

[1] Langley Formal Methods Program — NASA PVS Library.
`https://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library`

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

[3] Algoritmo Merge Sort — Wikipédia
`https://en.wikipedia.org/wiki/Merge_sort`