José Carlos de Oliveira Júnior

Verificando a corretude dos algoritmos de ordenação com o assistente de provas Coq

Belo Horizonte

5 de dezembro de 2019

José Carlos de Oliveira Júnior

Verificando a corretude dos algoritmos de ordenação com o assistente de provas Coq

Artigo de Projeto Orientado em Computação do tipo pesquisa científica elaborado como requisito para conclusão do curso de Ciência da Computação da Universidade Federal de Minas Gerais.

Universidade Federal de Minas Gerais

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Orientador: Mario Sérgio Alvim

Belo Horizonte

5 de dezembro de 2019

Resumo

Verificar a corretude de um software ou hardware tem sido de grande interesse para aplicações que precisam ser à prova de falhas. Sistemas de missão-crítica, quando apresentam alguma falha, podem causar perdas financeiras ou ter consequências fatais. Por isso, a aplicação de ferramentas de verificação de sistemas tem sido usada para desenvolver sistemas com alta garantia de sucesso. Uma ferramenta para escrever definições e demonstrações formais verificadas por máquina é o assistente de provas. Como parte final do meu Projeto Orientado em Computação, o assistente de provas Coq foi utilizado para demonstrar a corretude dos algoritmos dos algoritmos de ordenação por inserção e ordenação. Um algoritmo de ordenação está correto se sua saída é uma permutação da entrada e está ordenada. Esta definição foi escrita em Coq e foi aplicada nos algoritmos de ordenação citados para escrever suas demonstrações. O código das demonstrações encontra-se no apêndice deste trabalho. Apesar do Coq verificar se as demonstrações estão corretas, não é possível verificar por máquina se a especificação dos conceitos ou algoritmos estão corretos. Algumas novas táticas foram utilizadas em relação à primeira parte deste trabalho. É possível, por exemplo, criar novas táticas a partir das antigas. Uma possível continuação desse trabalho é a demonstração da corretude de outros algoritmos de ordenação.

Palavras-chave: Coq, corretude, ordenação, assistente de provas

Sumário

1 Introdução

Verificar a corretude de um software ou hardware tem sido de grande interesse para aplicações que precisam ser a prova de falhas. Sistemas de missão-crítica, quando apresentam alguma falha, podem causar perdas financeiras ou ter consequências fatais. Testes não são suficientes para garantir que falhas não irão acontecer, pois não verificará casos que possivelmente não estão cobertos. Por isto, a aplicação de ferramentas de verificação de sistemas, como assistente de provas, têm sido usadas para desenvolver sistemas com alta garantia de sucesso. (CLARKE et al., 2011)

O Coq é um assistente de provas aberto e desenvolvido no INRIA — *Institut National de Recherche en Informatique et en Automatique* — que provê uma linguagem formal para escrever definições matemáticas, algoritmos e teoremas em um ambiente semi-interativo capaz de checar provas. Uma ferramenta como o Coq é utilizada para verificar corretude de algoritmos e formalizar definições matemáticas. (INRIA)

Esta é a segunda parte do meu Projeto Orientado em Computação. A primeira parte, *Formalização e verificação com o assistente de provas Coq*, apresentou o assistente de provas Coq, sua IDE, conceitos de programação funcional, definições de funções e teoremas, aplicações de táticas de prova e aplicações de lógica.

Esta parte tem como objetivo formalizar e verificar a corretude de algoritmos de ordenação por inserção e seleção em lista de números naturais usando o assistente de provas Coq. O Capítulo 3 apresenta definição de permutação, ordenação e algoritmo de ordenação. É importante definir um algoritmo de ordenação, pois é pela definição que verificaremos a corretude da ordenação por inserção e seleção. O Capítulo 4 e o Capítulo 5 definem os algoritmos de ordenação e as demonstrações que foram realizadas em Coq para verificar os algoritmos. Por fim, o Capítulo 6 conclui o trabalho com comentários sobre corretude, dificuldade e escrita de especificações. O código-fonte em Coq do trabalho encontra-se no Apêndice A.

As definições e especificações foram baseados no texto e exercícios do livro *Verified Functional Algorithms* do Andrew W. Appel 2018.

2 Referencial Teórico

O processo de construção de um sistema que sempre retorna a resposta correta pode ser um trabalho árduo. Existem linguagens de programação que foram criadas com o objetivo de descrever o sistema matematicamente, bem como utilizar-la para verificar propriedades sobre este sistema. (PIERCE et al., 2019)

Para verificar se uma proposição é verdadeira, precisamos escrever uma demonstração utilizando argumentos lógicos. Para ajudar a escreve-las, ferramentas de auxílio foram criadas e se dividem em duas categorias. A primeira é de provadores automáticos. Nestes provadores, deve-se modelar o problema, inserir uma proposição e o provador retorna verdadeiro ou falso. Dependendo do problema, a resposta pode levar muito tempo para ser produzida. A segunda categoria é de assistentes de prova. Estes possuem funções automatizadas para proposições mais simples, mas necessitam de orientação humana para proposições mais complexas de serem provadas. (PIERCE et al., 2019)

O assistente de provas Coq é um resultado de cerca de 30 anos de pesquisa no INRIA. Com esta linguagem, é possível definir teoremas matemáticos e especificações de software, desenvolver demonstrações formais interativamente e realizar checagem por máquina destas demonstrações por um "kernel" relativamente pequeno. (INRIA)

O Coq utiliza paradigma de programação funcional. O princípio mais básico da programação funcional é que o cálculo é puro e não há efeitos colaterais. Uma vantagem do Coq é que a programação funcional serve como uma ponte entre a lógica e a ciência da computação. Com uma combinação de linguagem de programação funcional e ferramentas para afirmar e demonstrar afirmações lógicas, provas no Coq são programas. (PIERCE et al., 2019)

O assistente de provas Coq pode se usado para verificação de programas de diferentes maneiras. Uma delas é expressar "*o programa p está correto*", porém é preciso expressar matematicamente o que significa "correto" para o programa p. Também é preciso expressar matematicamente a semântica do programa. (PAULIN-MOHRING, 2011)

3 Corretude em algoritmos de ordenação

Neste capítulo, vamos descrever o que significa dizer "*o programa p está correto*" para a corretude de algoritmos de correção.

3.1 Lista ordenada

Um algoritmo de ordenação é uma função que recebe uma lista e retorna uma lista. Em Coq, é do tipo `list nat -> list nat`. Para toda menção de lista nesse trabalho, vamos considerar uma lista de números naturais. A primeira propriedade que queremos garantir é que a lista de saída seja ordenada.

Definição 3.1. Uma lista é ordenada nos seguintes casos:

1. Uma lista vazia é ordenada.
2. Uma lista com um elemento é ordenada.
3. Para todos os números naturais `x` e `y` e lista `l`. Se `x` $\leq$ `y` e a lista `y :: l` é ordenada, então a lista `x :: y :: l` também é ordenada.

O operador "`::`" acima é a função de concatenação de um elemento com uma lista. —

```
Inductive sorted: list nat -> Prop :=
  | sorted_nil : sorted nil
  | sorted_1 : forall (x : nat), sorted (x :: nil)
  | sorted_cons : forall (x y : nat) (l : list nat),
      x <= y -> sorted (y :: l) -> sorted (x :: y :: l).
```

Código 3.1 – Definição de lista ordenada em Coq

No Código 3.1, a função `sorted` é uma função que recebe uma lista e retorna uma proposição dizendo a lista é ordenada. Seus construtores são usados como teoremas para ser possível afirmar `sorted` para qualquer lista que se aplica.

3.2 Permutação entre listas

Apenas garantir que a saída seja uma lista ordenada não é suficiente. Seja `f` uma função que sempre retorna `[1; 2; 3]`. A saída é ordenada, mas não é necessariamente

uma permutação da lista de entrada. Então, definimos a segunda propriedade para um algoritmo de ordenação.

Definição 3.2. Seja `A` um conjunto. Sejam `x` e `y` elementos do conjunto `A`. Sejam `l`, `l'` e `l''` listas de `A`. São permutações de listas nos seguintes casos:

1. Uma lista vazia é permutação de uma lista vazia.
2. Se `l` é permutação de `l'`, então `x :: l` é permutação de `x :: l'`.
3. `x :: y :: l` é permutação de `y :: x :: l`.
4. Se `l` é permutação de `l'` e `l'` é permutação de `l''`, então `l` é permutação de `l''`.

A definição de permutação para qualquer tipo `A` faz parte da biblioteca padrão do Coq e algumas propriedades já demonstradas estão disponíveis para uso.

Essas duas propriedades são suficientes para definir um algoritmo de ordenação.

Definição 3.3. Uma função `f`, que recebe e retorna uma lista de números naturais, é um algoritmo de ordenação se, para toda lista `l`, `f l` é permutação de `l` (Definição 3.2) e é ordenada (Definição 3.1). —

4 Ordenação por inserção

Antes de mostrar o algoritmo de ordenação por inserção, vamos ver o passo de uma única inserção do algoritmo.

```
Fixpoint insert (i : nat) (l : list nat) :=
  match l with
  | nil => i::nil
  | h :: t => if i <=? h then i :: h :: t else h :: insert i t
 end.
```

Código 4.1 – Função de inserção ordenada em Coq

A função no Código 4.1 é uma função recursiva que recebe um número natural i e uma lista l e navega na lista da esquerda para a direita procurando um valor h tal que $i \leq h$. Ao encontrar, insere i antes de h e retorna a nova lista. Caso a lista seja vazia, retornará o único elemento inserido.

```
Fixpoint insertion_sort (l : list nat) : list nat :=
  match l with
  | nil => nil
  | h :: t => insert h (insertion_sort t)
  end.
```

Código 4.2 – Algoritmo de ordenação em Coq

A função `insertion_sort` remove todos os elementos da lista da esquerda para a direita e os insere da direita para a esquerda com a função `insert`.

```
insertion_sort [15; 4; 43]
    = insert 15 (insertion_sort [4; 43])
    = insert 15 (insert 4 (insertion_sort [43]))
    = insert 15 (insert 4 (insert 43 (insertion_sort [])))
    = insert 15 (insert 4 (insert 43 []))
    = insert 15 (insert 4 [43])
    = insert 15 [4; 43]
    = [4; 15; 43]
```

Com a definição do Código 4.2, sua corretude foi verificada e os seguintes teoremas foram demonstrados. O código das demonstrações se encontram no Apêndice A deste trabalho.

Lema 4.1. *Seja* `x` *um número natural. Seja* `l` *uma lista.* `insert x l` *é permutação de* `x :: l`.

```
Lemma insert_perm:
  forall (x : nat) (l : list nat), Permutation (x::l) (insert x l).
```

Teorema 4.1. *Seja* `l` *uma lista.* `l` *é permutação de* `insertion_sort l` *(Definição 3.2).*

```
Theorem sort_perm:
  forall (l : list nat), Permutation l (insertion_sort l).
```

Lema 4.2. *Seja* `a` *um número natural. Seja* `l` *uma lista. Se* `l` *é ordenada, então* `insert a l` *é ordenada.*

```
Lemma insert_sorted:
  forall (a : nat) (l : list nat), sorted l -> sorted (insert a l).
```

Teorema 4.2. *Seja* `l` *uma lista.* `insertion_sort l` *é ordenada (Definição 3.1).*

```
Theorem sort_sorted:
  forall (l : list nat), sorted (insertion_sort l).
```

Teorema 4.3. `insertion_sort` *é um algoritmo de ordenação. Seja* `l` *uma lista.* `insertion_sort l` *é permutação de* `l` *e é ordenada (Definição 3.3).*

```
Definition insertion_sort_correct : Prop :=
  is_a_sorting_algorithm insertion_sort.

Theorem insertion_sort_is_correct:
  insertion_sort_correct.
Proof.
  split.
  apply sort_perm.
  apply sort_sorted.
Qed.
```

5 Ordenação por seleção

Antes de mostrar o algoritmo de ordenação por seleção, vamos ver o passo de selecionar o menor elemento de uma lista e removê-lo.

```
Fixpoint select (x : nat) (l : list nat) : nat * list nat :=
  match l with
  |  nil => (x, nil)
  |  h :: t => if x <=? h
                 then let (j, l') := select x t in (j, h :: l')
                 else let (j, l') := select h t in (j, x :: l')
  end.
```

Código 5.1 – Função de seleção em Coq

A função do Código 5.1 recebe um número natural e uma lista e retorna um par com um número natural e uma lista. Apesar do objetivo ser retornar o menor elemento da lista, o parâmetro `x` é usado para ser o elemento padrão da saída. A função iniciará com `x` sendo o menor elemento. O par ordenado conterá o menor elemento entre `x` e os elementos da lista; e a lista da entrada removido o menor elemento.

```
Fixpoint selsort (l : list nat) (n : nat) {struct n} :=
  match l, n with
  | x :: r, S n' => let (y, r') := select x r
                    in y :: selsort r' n'
  | nil, _ => nil
  | _ :: _, O => nil
  end.
```

Código 5.2 – Função de ordenação por seleção em coq

A função de ordenação por seleção no Código 5.2 irá chamar a função `select` varias vezes de forma que a lista seja reconstruída de forma ordenada enquanto a lista original tem seus elementos removidos. A primeira chamada da função `select` tomará o primeiro elemento da lista como elemento padrão e fará comparações com o restante da lista.

```
selsort [15; 4; 43] 3
    = let (y, r') := select 15 [4; 43] in y :: selsort r' 2
    = 4 :: (selsort [15; 43])
    = 4 :: (let (y, r') := select 15 [43] in y :: selsort r' 1)
    = 4 :: 15 :: (selsort [43] 1)
    = 4 :: 15 :: (let (y, r') := select 43 [] in y :: selsort r' 0)
    = 4 :: 15 :: 45 :: selsort nil 0
```

```
= 4 :: 15 :: 45 :: nil
= [4; 15; 45]
```

O parâmetro `n` da função `selsort` deve ser o tamanho da lista. O motivo disso é que o Coq não consegue sozinho saber se a função `selsort`, que é recursiva, para em algum momento. Nesse caso, o Coq retorna um erro. Em Coq, uma lista é implementada usando um construtor que representa uma lista vazia e um construtor que concatena um elemento a uma lista. Na função `insertion_sort`, é garantido que irá parar, porque a recursão é feita no parâmetro `l`. A função para quando o parâmetro `l`, após remover os construtores de concatenação, chega no construtor de lista vazia. A função `selsort` não utiliza os construtores de lista, mas cria uma nova.

Existem duas formas de resolver este problema. Uma é provar para o Coq que a função para. Outra, que é usada aqui, é adicionar um parâmetro na qual será baseada a recursão da função. O parâmetro `n`, que é um número natural, é usado para ser o tamanho da lista e será decrementado a cada chamada de `selsort`. O tipo `nat` para números naturais em Coq usa o construtor `O` para representar o 0 e `S` para seus sucessores. A função para, pois removendo chamadas do construtor `S`, o construtor `O` eventualmente fará a função `selsort` retornar uma lista vazia.

Como `n` é sempre o tamanho da lista, podemos escrever a função `selection_sort` abaixo para ocultar esse parâmetro.

```
Definition selection_sort (l : list nat) :=
  selsort l (length l).
```

Código 5.3 – Função de ordenação por seleção em Coq usando o tamanho da lista

Com a definição do Código 5.3, sua corretude foi verificada e os seguintes teoremas foram demonstrados. O código das demonstrações se encontram no Apêndice A deste trabalho.

Lema 5.1. *Sejam* `l`, `l'` *e* `r` *listas. Sejam* `x` *e* `r` *números naturais. Seja* `select x l` = (`y`, `r`). `y` :: `r` *é permutação de* `x` :: `l`.

```
Lemma select_perm:
  forall (x : nat) (l : list nat),
    let (y, r) := select x l in Permutation (x :: l) (y :: r).
```

Teorema 5.1. *Sejam* `l` *uma lista.* `selection_sort l` *é permutação de* `l` *(Definição 3.2).*

```
Theorem selection_sort_perm:
  forall (l : list nat), Permutation l (selection_sort l).
```

Teorema 5.2. *Sejam* `al` *e* `bl` *listas. Sejam* `x` *e* `y` *números naturais. Se* `select x al` = (`y`, `bl`), *então para todo natural* `z` *em* `bl`, `y` $\leq$ `z`.

```
Theorem select_smallest:
  forall (x : nat) (al : list nat) (y : nat) (bl : list nat),
    select x al = (y,bl) ->
    Forall (fun z => y <= z) bl.
```

Teorema 5.3. *Seja* `al` *uma lista.* `selection_sort al` *é ordenada (Definição 3.1).*

```
Theorem selection_sort_sorted:
  forall (al : list nat), sorted (selection_sort al).
```

Teorema 5.4. `selection_sort` *é um algoritmo de ordenação. Seja* `l` *uma lista.* `selection_sort l` *é permutação de* `l` *e é ordenada (Definição 3.3).*

```
Definition insertion_sort_correct : Prop :=
  is_a_sorting_algorithm selection_sort.

Theorem selection_sort_is_correct:
  selection_sort_correct.
Proof.
  split.
  apply selection_sort_perm.
  apply selection_sort_sorted.
Qed.
```

6 Conclusão

Apesar das provas serem necessárias para verificar a corretude de algoritmos de ordenação, a parte mais importante deste trabalho é escrever a proposição que deverá ser provada. Em outras palavras, devemos escrever quais propriedades um sistema deve satisfazer para ser considerado correto. Apesar do Coq verificar as provas, não é possível verificar se a especificação está correta dado um problema (APPEL, 2018). Nesse trabalho, é suficiente dizer um algoritmo de ordenação deve satisfazer a Definição 3.1 e a Definição 3.2.

A forma como o programa e as especificações são escritas tem um papel importante no desenvolvimento das provas. O Coq é um assistente de provas e diferentes maneiras de escrever o programa e as especificações podem levar a diferentes níveis de dificuldade para o programador provar sua corretude.

Em relação a primeira parte deste trabalho, que foi focada no aprendizado por tópicos direcionados sobre Coq, qualquer recurso poderia ser usado para escrever as provas dos teoremas. Algumas técnicas novas foram apresentadas. A tática `omega` foi usada para provar teoremas com equações e inequações. É possível usar `Ltac` para criar novas táticas a partir das táticas básicas como forma de resumir provas que utilizam uma mesma sequência de táticas várias vezes.

Por fim, o objetivo deste trabalho foi concluído com a verificação dos algoritmos de ordenação por inserção e seleção. Uma possível continuação deste trabalho a aplicação destas definições para outros algoritmos de ordenação como *quick sort* e *merge sort*.

Referências

APPEL, A. W. *Verified Functional Algorithms*. [S.l.]: Electronic textbook, 2018. (Software Foundations series, volume 3). Version 1.4. https://softwarefoundations.cis.upenn.edu/vfa-current/index.html. Citado 2 vezes nas páginas 4 e 13.

CLARKE, E. M. et al. Model checking and the state explosion problem. In: SPRINGER. *LASER Summer School on Software Engineering*. [S.l.], 2011. p. 1–30. Citado na página 4.

INRIA. *The Coq Proof Assistant*. 2019. Disponível em: <https://coq.inria.fr/>. Acesso em: 3 abr. 2019. Citado 2 vezes nas páginas 4 e 5.

PAULIN-MOHRING, C. Introduction to the coq proof-assistant for practical software verification. In: SPRINGER. *LASER Summer School on Software Engineering*. [S.l.], 2011. p. 45–95. Citado na página 5.

PIERCE, B. C. et al. *Logical Foundations*. [S.l.]: Electronic textbook, 2019. (Software Foundations series, volume 1). Version 5.6. https://softwarefoundations.cis.upenn.edu/lf-current/index.html. Citado na página 5.

APÊNDICE A – Código-fonte

A.1 Utils.v

```
Require Export Coq.omega.Omega.
Require Export Coq.Bool.Bool.

Lemma beq_reflect:
  forall (x y : nat), reflect (x = y) (x =? y).
Proof.
  intros x y.
  apply iff_reflect.
  symmetry.
  apply beq_nat_true_iff.
Qed.

Lemma blt_reflect:
  forall (x y : nat), reflect (x < y) (x <? y).
Proof.
  intros x y.
  apply iff_reflect.
  symmetry.
  apply Nat.ltb_lt.
Qed.

Lemma ble_reflect:
  forall (x y : nat), reflect (x <= y) (x <=? y).
Proof.
  intros x y.
  apply iff_reflect.
  symmetry.
  apply Nat.leb_le.
Qed.

Hint Resolve blt_reflect ble_reflect beq_reflect : bdestruct.

Ltac bdestruct X :=
  let
    H := fresh
```

```
  in let
    e := fresh "e"
  in
   evar (e: Prop);
   assert (H: reflect e X);
   subst e;
    [eauto with bdestruct
    | destruct H as [H|H];
       [ | try first [apply not_lt in H | apply not_le in H]]].

Ltac inv H := inversion H; clear H; subst.
```

A.2 Sorted.v

```
Require Export Coq.Lists.List.
Require Export Permutation.
From sorting Require Export Utils.

Inductive sorted: list nat -> Prop :=
  | sorted_nil : sorted nil
  | sorted_1 : forall (x : nat), sorted (x :: nil)
  | sorted_cons : forall (x y : nat) (l : list nat),
      x <= y -> sorted (y :: l) -> sorted (x :: y :: l).

Definition is_a_sorting_algorithm (f: list nat -> list nat) :=
  forall (al : list nat), Permutation al (f al) /\ sorted (f al).
```

A.3 InsertionSort.v

```
From sorting Require Export Utils.
From sorting Require Export Sorted.


(** * Definition *)

Fixpoint insert (i : nat) (l : list nat) :=
  match l with
  | nil => i::nil
  | h :: t => if i <=? h then i :: h :: t else h :: insert i t
 end.

Fixpoint insertion_sort (l : list nat) : list nat :=
```

```
  match l with
  | nil => nil
  | h :: t => insert h (insertion_sort t)
  end.


(** * Correctness goal **)

Definition insertion_sort_correct : Prop :=
  is_a_sorting_algorithm insertion_sort.


(** * Permutations **)

Lemma insert_perm:
  forall (x : nat) (l : list nat), Permutation (x::l) (insert x l).
Proof.
  induction l. {
    auto.
  } {
    simpl.
    bdestruct (x <=? a). {
      auto.
    } {
      induction l. {
        simpl.
        apply perm_swap.
      } {
        apply perm_trans with (a :: x :: a0 :: l).
        apply perm_swap.
        apply perm_skip.
        apply IHl.
      }
    }
  }
Qed.

Theorem sort_perm:
  forall (l : list nat), Permutation l (insertion_sort l).
Proof.
  intros.
  induction l. {
```

```
    auto.
  } {
    simpl.
    induction IHl. {
      auto.
    } {
      simpl.
      bdestruct (a <=? x). {
        auto.
      } {
        apply perm_trans with (x :: a :: l).
        apply perm_swap.
        auto.
      }
    } {
      simpl.
      bdestruct (a <=? y);
      bdestruct (a <=? x). {
        apply perm_skip.
        apply perm_swap.
      } {
        apply perm_trans with (a :: x :: y :: l).
        apply perm_skip.
        apply perm_swap.
        apply perm_swap.
      } {
        apply perm_skip.
        apply perm_swap.
      } {
        apply perm_trans with (a :: x :: y :: l).
        apply perm_skip.
        apply perm_swap.
        apply perm_trans with (x :: a :: y :: l).
        apply perm_swap.
        apply perm_skip.
        apply perm_trans with (y :: a :: l).
        apply perm_swap.
        apply perm_skip.
        apply insert_perm.
      }
    } {
      assert (Permutation (a :: l') (insert a l')).
```

```
      apply insert_perm.
      assert (Permutation (a :: l) (a :: l')).
      apply perm_skip.
      apply IHl1.
      apply perm_trans with (a :: l');
      auto.
    }
  }
Qed.


(** * Insertion of a element in a sorted list **)

Lemma insert_sorted:
  forall (a : nat) (l : list nat), sorted l -> sorted (insert a l).
Proof.
  intros.
  induction H. {
   simpl.
   apply sorted_1.
  } {
    simpl.
    bdestruct (a <=? x). {
      apply sorted_cons.
      trivial.
      apply sorted_1.
    } {
      apply sorted_cons.
      omega.
      apply sorted_1.
    }
  } {
    simpl.
    bdestruct (a <=? x);
    bdestruct (a <=? y);
    apply sorted_cons;
    try omega;
    try apply sorted_cons;
    try omega;
    try apply H0.
    simpl in IHsorted.
    bdestruct (a <=? y). {
```

```
      omega.
    } {
      trivial.
    }
  }
Qed.


(** * A list applied with insertion sort is sorted **)

Theorem sort_sorted:
  forall (l : list nat), sorted (insertion_sort l).
Proof.
  intros.
  induction l. {
    simpl.
    apply sorted_nil.
  } {
    simpl.
    apply insert_sorted.
    trivial.
  }
Qed.


(** * Wrapping up **)

Theorem insertion_sort_is_correct:
  insertion_sort_correct.
Proof.
  split.
  apply sort_perm.
  apply sort_sorted.
Qed.
```

A.4 SelectionSort.v

```
From sorting Require Export Utils.
From sorting Require Export Sorted.


(** * Definition  *)
```

```
Fixpoint select (x : nat) (l : list nat) : nat * list nat :=
  match l with
  |  nil => (x, nil)
  |  h :: t => if x <=? h
                then let (j, l') := select x t in (j, h :: l')
                else let (j, l') := select h t in (j, x :: l')
  end.

Fixpoint selsort (l : list nat) (n : nat) {struct n} :=
  match l, n with
  | x :: r, S n' => let (y,r') := select x r
                in y :: selsort r' n'
  | nil, _ => nil
  | _ :: _, O => nil
  end.

Definition selection_sort (l : list nat) :=
  selsort l (length l).


(** * Correctness goal  *)

Definition selection_sort_correct : Prop :=
  is_a_sorting_algorithm selection_sort.


(** * Permutations *)

Lemma select_perm:
  forall (x : nat) (l : list nat),
    let (y, r) := select x l in Permutation (x :: l) (y :: r).
Proof.
  intros x l; revert x.
  induction l; intros; simpl in *. {
    apply Permutation_refl.
  } {
    unfold select.
    bdestruct (x <=? a); fold select. {
      specialize (IHl x).
      destruct (select x l) eqn:Seq.
      apply perm_trans with (a :: n :: l0). {
```

```
        apply Permutation_sym.
        apply perm_trans with (a :: x :: l). {
          now apply perm_skip.
        } {
          apply perm_swap.
        }
      } {
        apply perm_swap.
      }
    } {
      specialize (IHl a).
      destruct (select a l) eqn:Seq.
      apply perm_trans with (x :: n :: l0). {
        now apply perm_skip.
      } {
        apply perm_swap.
      }
    }
  }
Qed.

Lemma selsort_perm:
  forall (n : nat) (l : list nat), length l = n -> Permutation l (selsort l n)
Proof.
  induction n. {
    intros.
    destruct l. {
      apply perm_nil.
    } {
      inversion H.
    }
  } {
    intros.
    destruct l. {
      intros.
      inversion H.
    } {
      simpl.
      destruct (select n0 l) eqn:Seq.
      apply perm_trans with (n1 :: l0). {
        assert (let (y, r) := select n0 l in Permutation (n0 :: l) (y :: r)).
          apply (select_perm n0 l).
```

```
        }
        destruct (select n0 l).
        inv Seq.
        apply H0.
      } {
        apply perm_skip.
        apply IHn.
        assert (length (n0::l) = (length (n1::l0))). {
          apply Permutation_length.
          assert (let (y, r) := select n0 l in Permutation (n0 :: l) (y :: r))
            apply (select_perm n0 l).
          }
          destruct (select n0 l).
          inv Seq.
          apply H0.
        }
        inv H0.
        inv H.
        reflexivity.
      }
    }
  }
Qed.

Theorem selection_sort_perm:
  forall (l : list nat), Permutation l (selection_sort l).
Proof.
  unfold selection_sort.
  intros.
  apply selsort_perm.
  reflexivity.
Qed.


(** * [select] selects the smallest element of a list *)

Lemma select_smallest_aux:
  forall (x : nat) (al : list nat) (y : nat) (bl : list nat),
    Forall (fun z => y <= z) bl ->
    select x al = (y,bl) ->
    y <= x.
Proof.
```

```
  intros.
  pose (select_perm x al).
  rewrite H0 in y0.
  apply (Permutation_in x) in y0. {
    destruct y0. {
      omega.
    } {
      rewrite Forall_forall in H.
      apply H.
      apply H1.
    }
  } {
    apply in_eq.
  }
Qed.

Theorem select_smallest:
  forall (x : nat) (al : list nat) (y : nat) (bl : list nat),
    select x al = (y,bl) ->
    Forall (fun z => y <= z) bl.
Proof.
  intros x al.
  revert x.
  induction al; intros; simpl in *. {
    inv H.
    easy.
  } {
    bdestruct (x <=? a). {
      destruct select eqn:Seq.
      inv H.
      apply Forall_cons. {
        apply (le_trans y x a). {
          apply (select_smallest_aux x al y l). {
            apply (IHal x y l).
            easy.
          } {
            easy.
          }
        } {
          easy.
        }
      } {
```

```
        apply (IHal x y l).
        easy.
      }
    } {
      destruct (select a al) eqn:?H.
      inv H.
      apply Forall_cons. {
        assert (y <= a). {
          apply (select_smallest_aux a al y l). {
            apply (IHal a y l).
            apply H1.
          } {
            apply H1.
          }
        }
        omega.
      } {
        apply (IHal a y l).
        easy.
      }
    }
  }
Qed.


(** * A list applied with selection sort is sorted **)

Lemma selection_sort_sorted_aux:
  forall (y : nat) (bl : list nat),
   sorted (selsort bl (length bl)) ->
   Forall (fun z : nat => y <= z) bl ->
   sorted (y :: selsort bl (length bl)).
Proof.
  induction bl. {
    simpl.
    intros.
    apply sorted_1.
  } {
    intros.
    simpl in *.
    destruct select eqn:Seq.
    apply sorted_cons. {
```

```
      rewrite Forall_forall in H0.
      apply H0.
      apply Permutation_in with (n :: l). {
        pose (select_perm a bl).
        rewrite Seq in y0.
        apply Permutation_sym.
        apply y0.
      } {
        apply in_eq.
      }
    } {
      apply H.
    }
  }
Qed.

Theorem selection_sort_sorted:
  forall (al : list nat), sorted (selection_sort al).
Proof.
  intros.
  unfold selection_sort.
  remember (length al) as n.
  generalize dependent al.
  induction n. {
    intros.
    simpl.
    destruct al. {
      apply sorted_nil.
    } {
      apply sorted_nil.
    }
  } {
    intros.
    destruct al. {
      apply sorted_nil.
    } {
      unfold selsort.
      fold selsort.
      destruct (select n0 al) eqn:Seq.
      pose (select_perm n0 al).
      rewrite Seq in y.
      apply Permutation_length in y.
```

```
      rewrite <- Heqn in y.
      inversion y.
      apply (selection_sort_sorted_aux n1 l). {
        rewrite <- H0.
        apply IHn.
        apply H0.
      } {
        apply (select_smallest n0 al n1 l).
        apply Seq.
      }
    }
  }
Qed.


(** * Wrapping up **)

Theorem selection_sort_is_correct:
  selection_sort_correct.
Proof.
  split.
  apply selection_sort_perm.
  apply selection_sort_sorted.
Qed.
```