# Insertion Sort: A Case Study in Rocq

Maria João Frade

HASLab - INESC TEC
Departamento de Informática, Universidade do Minho

2025/2026



## Sorting a list

A sorting algorithm for a list must rearrange its elements into a totally ordered list.

- Therefore, the output of a sorting algorithm must be a sorted list that is a permutation of the input list.

This case study is devoted to a well-known algorithm–**Insertion Sort**–which consists of iteratively applying an insert function that inserts an element into a sorted list. For simplicity, we focus on sorting lists of integers.

Load the file `insertionSort.v` in the Rocq Prover to follow the examples of the coming slides. Analyse the examples and solve the exercises proposed.



## Sorting a list

A simple characterisation of sorted lists consists in requiring that two consecutive elements be compatible with the $\leq$ relation.

```
Notation "x == y" := (Z.eq_dec x y) (at level 70, no associativity).
Notation "x <?? y" := (Z_lt_ge_dec x y) (at level 70, no associativity).

Set Implicit Arguments.
Open Scope Z_scope.
```

We can codify this with the following inductive predicate:

```
Inductive Sorted : list Z -> Prop :=
  | sorted0 : Sorted nil
  | sorted1 : forall z:Z, Sorted (z :: nil)
  | sorted2 : forall (z1 z2:Z) (l:list Z),
          z1 <= z2 -> Sorted (z2 :: l) -> Sorted (z1 :: z2 :: l).
```



## Sorting a list

To capture permutations, rather than using an inductive definition, we define the relation via an auxiliary function that counts the number of occurrences of an element in a list.

```
Fixpoint count (z:Z) (l:list Z) {struct l} : nat :=
  match l with
  | nil => 0%nat
  | z' :: l' => if z == z' then S (count z l')
                           else count z l'
  end.
```

The predicate to check list permutation can now be definied as

```
Definition Perm (l1 l2:list Z) : Prop :=
                            forall z, count z l1 = count z l2.
```

## Permutation properties

Perm is an equivalence relation:

```
Lemma Perm_reflex : forall l:list Z, Perm l l.
Lemma Perm_sym : forall l1 l2, Perm l1 l2 -> Perm l2 l1.
Lemma Perm_trans : forall l1 l2 l3,
                        Perm l1 l2 -> Perm l2 l3 -> Perm l1 l3.
```

Other usefull lemmas:

```
Lemma Perm_cons : forall a l1 l2,
                             Perm l1 l2 -> Perm (a::l1) (a::l2).
Lemma Perm_cons_cons : forall x y l, Perm (x::y::l) (y::x::l).
```

**Exercise:**
Complete the missing proofs.



## Defining the isort function

First, we define a the insert function that inserts an element in a sorted list.

```
Fixpoint insert (x:Z) (l:list Z) {struct l} : list Z :=
  match l with
  | nil => x :: nil
  | h :: t => if (x <?? h) then x :: (h :: t)
                          else h :: (insert x t)
  end.
```

Insertion sort is defined by the iterative application of the insert function.

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



## Proving isort correcteness

isort correcteness can be codified in the following theorem:

```
Theorem isort_correct : forall (l l':list Z),
                              l'=isort l -> Perm l l' /\ Sorted l'.
```

We will certainly need auxiliary lemmas... Let us make a prospective proof attempt:

```
Proof.
  induction l; intros.
  - unfold Perm; rewrite H; split; auto. simpl. constructor.
  - simpl in H. rewrite H. (* ?????????? *)
    pose proof (IHl (isort l)) as H1. specialize (H1 eq_refl).
```

```
a : Z
l : list Z
IHl : forall l' : list Z, l' = isort l -> Perm l l' /\ Sorted l'
l' : list Z
H : l' = insert a (isort l)
H1 : Perm l (isort l) /\ Sorted (isort l)
====================================================================
Perm (a :: l) (insert a (isort l)) /\ Sorted (insert a (isort l))
```



## Lemmata

It is now clear what are the lemmas we need.

```
Lemma insert_Perm : forall x l, Perm (x::l) (insert x l).

Lemma insert_Sorted : forall x l, Sorted l -> Sorted (insert x l).
```

In order to prove them the following lemmas about count, may be useful.

```
Lemma count_insert_eq : forall x l,
                        count x (insert x l) = S (count x l).

Lemma count_cons_diff : forall z x l,
                        z <> x -> count z l = count z  (x :: l).

Lemma count_insert_diff : forall z x l,
                         z <> x -> count z l = count z (insert x l).
```

**Exercise:**
Complete the missing proofs.

## Proving isort correcteness

Now we can conclude the proof of correctness.

**Exercise: Complete the proof.**

```
Theorem isort_correct : forall (l l':list Z),
                        l'=isort l -> Perm l l' /\ Sorted l'.
Proof.
  induction l; intros.
  - unfold Perm; rewrite H; split; auto. simpl. constructor.
  - simpl in H.
    rewrite H.                (* ??????????? *)
    destruct (IHl (isort l)).

    ...
```



## Using a strong specification type

An alternative approach is to use specification types ($\Sigma$-types) when defining the insertion sort function. This will force the prove that the specification is inhabited.

**Exercise: Complete the proof.**

```
Definition inssort : forall (l:list Z),
                               { l' | Perm l l' & Sorted l' }.
 induction l.
  - exists nil. constructor. constructor.
  - elim IHl. intros. exists (insert a x).
    ...
Defined.
```

The function inssort just defined contains a computational part that says how to sort the input list and a certificate of its correction (a proof that the output list is a permutation of the input list, and is sorted).

**Exercise:**

Use the extraction mechanism of the Rocq Prover to extract the computational content of inssort into an Haskell function.