# Insertion Sort Correctness Proofs in Lean: A Comparison of Different Specifications

Jein Ryu

November 28, 2025

**Abstract**

This study shows how different formulations of specifications influence the structure and difficulty of correctness proofs in Lean. Using insertion sort as a case study, we formalize two different formulations for each of the two correctness properties: sortedness and content preservation. For sortedness, we compare an inductive formulation aligned with Lean's proof style and an index-based formulation that more directly expresses the mathematical intuition. For content preservation, we compare a permutation-based specification with an algebraic multiset-based specification. Our results demonstrate that the choice of specification can influence both the structure and the difficulty of the proofs, highlighting the importance of specification formulation.

## 1 Introduction

Formal verification of an algorithm requires a formal specification describing its desired properties. However, a specification can be expressed in multiple ways, even within the same language. The goal of this research is to compare how different formulations of a specification affect the structure and complexity of correctness proofs. In particular, we analyze the correctness proof of insertion sort algorithm in Lean under different specifications. This work was inspired by *Verified Functional Algorithms* [1]. The entire source code of this research is available at this repository: `https://github.com/jeinryu/InsertionSort`.

## 2 Related Works

### 2.1 Formal Verification of Sorting Algorithms

The correctness proof of sorting algorithms has been studied in various proof assistants such as Coq, Isabelle/HOL and Lean. These formalizations typically follow a common pattern: they used an inductive predicate to express sortedness, and permutation to define content preservation.

Within the Lean ecosystem, the work of Silváši and Tomášek [2] presents a formalization of insertion sort, including proofs of stability and correctness. Their approach also relies on the inductive definition of sortedness and uses permutations to express content preservation. Lean is well suited for such developments due to its mature mathematical library `mathlib` and its expressive inductive types. While these prior works establish robust correctness proofs, they generally adopt a single formulation for each property.

### 2.2 Multiple Specifications of Sortedness and Content Preservation

On the other hand, a single mathematical property may admit several equivalent formal specifications. For example, *Software Foundations* [1] discusses alternative ways to define sortedness and content-preservation of lists. However, they formulate the proofs in Coq and do not explicitly compare how different specifications affect the correctness proofs.

This paper makes the following key contributions: our work investigates two different definitions of sortedness (an inductive definition and an index-based numerical definition) and two different definitions of content preservation (permutation and multiset equivalence), formalized in Lean. Beyond simply presenting these definitions and proofs, we analyze how they influence the proof structure and the overall proof complexity. To our knowledge, there is limited prior work that examines multiple specifications of the same property and analyzes differences.

# 3 Methods

The complete definitions and specifications are available in the file: Spec.lean

## 3.1 Definition of Insertion Sort Algorithm in Lean

Below is the Lean code for the insertion sort algorithm.

```
def insrt (n : Nat) (l : List Nat) : List Nat :=
  match l with
  | [] => [n]
  | h :: t => if n <= h then n :: l else h :: insrt n t

def sort (l : List Nat) : List Nat :=
  match l with
  | [] => []
  | h :: t => insrt h (sort t)
```

We first defined the `insrt` operation recursively, which inserts a new element into the input list. If the list is empty, then it returns a singleton list containing the new element. Else, it compares the new element $n$ with the head element $h$ of the list; if $n \leq h$, then $n$ is inserted in front of the input list. If not, then $n$ is recursively inserted into the tail of the list.

Then we defined the `sort` algorithm, which applies the insertion sort algorithm to the input list using the `insrt` operation recursively. If the list is empty, it returns an empty list. Else, it applies `sort` algorithm to its tail recursively, and then `insrt` the head element.

## 3.2 Formulations of Specifications

A correctness proof for a sorting algorithm typically involves two properties. First, the output list has to be sorted. Second, the output list should preserve the contents of the original list. This paper compares the correctness proofs for each property using two different specifications.

### 3.2.1 Sortedness: Inductive and Index-Based Formulation

**Inductive Definition** First, we define the property of sortedness with respect to the inductive definition of list in Lean, which is the inductive formulation used in most of the prior studies mentioned in section 2.1. The predicate `Sorted` is defined with three constructors: `nil, singleton, cons`.

- `Sorted.nil`: An empty list is `Sorted`.
- `Sorted.singleton n`: Any singleton list is `Sorted`.
- `Sorted.cons n m l`: A list with more than two elements is `Sorted` if its first two elements are in order and the tail of the list is also `Sorted`;
  `n ≤ m → Sorted (m :: l) → Sorted (n :: m :: l)`

**Index-Based Definition** In addition to the inductive definition, the property of sortedness can also be defined using list indices. This definition expresses a more direct and intuitive notion of sortedness: an element that appears earlier in the list must be less than or equal to any element that comes after it. The predicate `Sorted′` can be defined using the property `∀ (i j) i ≤ j → l[i] ≤ l[j]`. In the actual Lean implementation, one must be careful with regard to the index bounds, since Lean is strict about accessing out-of-bounds indices. To handle this, we use the `Option` type to safely represent element retrieval, ensuring that any access to the list returns `Some value` if the index is valid, or `none` otherwise.

However, one might still question whether these formal definitions fully capture the informal notion of sortedness as conceived in our minds. Although we cannot formally prove that our specifications completely correspond to our informal intuition, we can show their equivalence. Such equivalence provide more confidence for the consistency of our formulations. The equivalence proofs are presented in the following section.

### 3.2.2 Content-Preservation: Permutation and Multiset-Based Formulation

**Permutation-Based Formulation** We use the permutation relation defined in Lean. One can check the entire source code here. Permutation relation is defined inductively with four constructors: `nil`, `cons`, `swap`, `trans`.

- `perm.nil`: An empty list is a permutation of an empty list.
- `perm.cons`: If two lists are in permutation relation, then the lists obtained by `cons`-ing a same element are still in the permutation relation.
- `perm.swap`: A list obtained by swapping the first two elements is a permutation of the original list.
- `perm.trans`: A permutation relation is transitive, i.e. if $l_1$ is a permutation of $l_2$ and $l_2$ is a permutation of $l_3$, then $l_1$ is a permutation of $l_3$.

**Multiset-Based Formulation** A multiset is a data structure that allows multiple occurrences of the same element. We use multisets to represent the contents of a list. We implement the multiset operations `empty, singleton, union` by defining multiset as a function that counts the given element. Then we define the function `contents`, which counts the number of the given element in the given list. Later, we argue that `contents al = contents bl` is equivalent to `List.Perm al bl`.

```
def contents (l : List Nat) : Multiset :=
  match l with
  | [] => empty
  | x :: xs => union (singleton x) (contents xs)
```

# 4 Proof of Sortedness

## 4.1 Inductive Definition

The complete proof for this subsection is avaliable in the file: InductiveProof.lean.

To prove that `sort` algorithm satisfies the `Sorted` property, we can divide into two steps. First, we show that the `insrt` operation preserves the `Sorted` property: that is, if a list is already `Sorted`, then it remains `Sorted` after insertion. After that, we show that `sort` algorithm satisfies the `Sorted` property using the previous theorem. Below are the theorems needed for the entire proof.

`theorem insrt_Sorted (a : Nat) (l: List Nat) : Sorted l → Sorted (insrt a l)`

`theorem sort_Sorted (l : List Nat) : Sorted (sort l)`

We first prove `theorem insrt_Sorted`. Since the predicate `Sorted` is defined inductively, we can proceed by induction on the premise `(Sorted l)`. We want to prove `Sorted (insrt a l)`. Consider the three constructors of `Sorted` applied to the premise:

- `Sorted.nil`: Since `l` is empty, `insrt a l` is a singleton list. Thus the resulting list is still `Sorted`.
- `Sorted.singleton m`: Our goal is to show `Sorted (insrt a [m])`. We perform case analysis on whether `a` $\leq$ `m` holds.
  - if `a` $\leq$ `m`, then the resulting list is `a :: m`, which is `Sorted`.
  - if `a > m`, then the resulting list is `m :: a`, which is `Sorted`.
- `Sorted.cons x y l`: We have `x` $\leq$ `y`, `Sorted (y :: l)` as hypothesis. Our inductive hypothesis says `Sorted (y :: l)` $\rightarrow$ `Sorted (insrt a (y :: l))`, so we have `Sorted (insrt a (y :: l))`. Our goal is to show `Sorted (insrt a (x :: y :: l))`. We perform case analysis on `a` $\leq$ `x`.
  - if `a` $\leq$ `x`, then the resulting list is `a :: x :: y :: l`, which is `Sorted` by the hypothesis.
  - if `a > x`, then the resulting list is `x :: insrt a (y :: l)`. We perform case analysis on `a` $\leq$ `y`.
    - if `a` $\leq$ `y`, then the resulting list is `x :: a :: y :: l`, which is `Sorted` by the hypothesis and `Sorted.cons`.
    - if `a > y`, then the resulting list is `x :: y :: insrt a l`, which is `Sorted` by the hypothesis and `Sorted.cons`.

Using `theorem insrt_Sorted` together with induction on the input list, one can easily prove `theorem sort_Sorted`. The complete proofs in this section are available in the file: InductiveProof.lean.

## 4.2 Index-Based Definition

The complete proof for this subsection is avaliable in the file: IndexProof.lean.

To show that `insrt` preserves `Sorted'`, we need further reasoning. Since we have to reason about elements of the new list `insrt a l`, we should first show that any element in the resulting list either comes from the original list or is the newly inserted element. We also make use of the fact that the tail of a `Sorted'` list is also `Sorted'`.

```
theorem insrt_getElem? (l: List Nat) (a i iv : Nat) :
  (insrt a l)[i]? = some iv → a = iv ∨ ∃ i': Nat, l[i']? = some iv
theorem cons_Sorted' (x : Nat) (xs : List Nat) : Sorted' (x :: xs) → Sorted' xs
```

The above theorems are auxiliary lemmas for the proof. `theorem insrt_getElem?` states that an element of the inserted list is either the new element or contained in the original list, and `theorem cons_Sorted'` states that if a list is `Sorted'`, then its tail is also `Sorted'`. Their proofs are available in the file: IndexProof.lean.

To show that `insrt` preserves the `Sorted'` property, we perform induction on the input list `l`. Since the base case `List.nil` is simple, we omit the detail. In the inductive step `List.cons x xs`, we have `Sorted' xs → Sorted' (insrt a xs)` as the inductive hypothesis. Our goal is to show `Sorted' (x :: xs) → Sorted' (insrt a (x :: xs))`. We first split on whether the new element is inserted before or after `x`, i.e. `a ≤ x`, same as in the proof of `theorem insrt_Sorted`. Note that the elements of the resulting list are either from the original list or the new element, and they differ in proof strategy. Thus, we perform a case analysis on the index pairs `(i, j)` and check whether the statement holds for each case. Proving each case requires the above auxiliary lemmas and additional theorems about list indices in Lean since we have to deal with list index.

- `a ≤ x`: Our goal is `(a :: x :: xs)[i] ≤ (a :: x :: xs)[j]`, where `i ≤ j`. We perform case analysis on `(i, j)`.
  - `(0, 0)/ (i + 1, 0)`: Obvious/ Contradiction.
  - `(0, j + 1)`: Rewrite the goal as `a ≤ (x :: xs)[j]`, which is true by the premise `Sorted' (x :: xs)`.
  - `(i + 1, j + 1)`: Rewrite the goal as `(x :: xs)[i] ≤ (x :: xs)[j]`, which is true by the premise.
- `a > x`: Our goal is `(x :: insrt a xs)[i] ≤ (x :: insrt a xs)[j]`. Note that, by the premise `Sorted' (x :: xs)` and `theorem cons_Sorted'`, we get `Sorted' xs`. We again perform case analysis on `(i, j)`.
  - `(0, 0)/ (i + 1, 0)`: Obvious/ Contradiction.
  - `(0, j + 1)`: Rewrite the goal as `x ≤ (insrt a xs)[j]`. Here, we use `theorem insrt_getElem?`.
    - `(insrt a xs)[j]` is `a`: Obvious.
    - `(insrt a xs)[j]` is `xs[j']`: Note that `xs[j'] = (x :: xs)[j' + 1]`. Then we can use the premise `Sorted' (x :: xs)`.
  - `(i + 1, j + 1)`: Rewrite the goal as `(insrt a xs)[i] ≤ (insrt a xs)[j]`, which is true by the inductive hypothesis.

## 4.3 Equivalence of the Two Definitions

The complete proof for this subsection is avaliable in the file: SortedEq.lean.

We've seen two different definitions of sortedness, `Sorted` and `Sorted'`. However, how do we know that our definition is correct? We can't formally prove it, but proving their equivalence may provide stronger assurance that our formalization is consistent. To prove equivalence, we need to show both directions:

```
theorem Sorted_Sorted' (l : List Nat) : Sorted l → Sorted' l
theorem Sorted'_Sorted (l : List Nat) : Sorted' l → Sorted l
```

**From `Sorted` to `Sorted'`.** The proof proceeds by induction on the premise `Sorted l`. Then we can do case analysis on the index and utilize additional theorems about list indices in Lean.

- `Sorted.nil, Sorted.singleton`: Easy.
- `Sorted.cons n m l`: Our goal is to show `Sorted' (n :: m :: l)` from hypothesis `Sorted (m :: l)` and `Sorted' (m :: l)`. Introducing variable `i, j` for indices, we should show `(n :: m :: l)[i] ≤ (n :: m :: l)[j]`. We perform case analysis on `i`, and perform similar case analysis on `j`. One can easily prove each case.

**From `Sorted'` to `Sorted`.** The proof proceeds by induction on the input list. Case `nil` follows immediately, thus we omit the detail. For the inductive case `cons x xs`, our goal is to prove `Sorted (x :: xs)` from `Sorted' (x :: xs)`. We again apply induction on the tail list `xs`. Then we can apply the constructors for `Sorted`.

- `xs = nil`: Goal becomes `Sorted [x]`, which is true.
- `xs = cons y ys`: Goal becomes `Sorted (x :: y :: ys)`. By applying `Sorted.cons`, we get the following two subgoals:
  - `x ≤ y`: Follows by the original premise `Sorted' (x :: y :: ys)`
  - `Sorted (y :: ys)`: Applying the inductive hypothesis, it suffices to show `Sorted' (y :: ys)`, which follows by the original premise `Sorted' (x :: y :: ys)` and `theorem cons_Sorted'`.

# 5 Proof of Content Preservation

## 5.1 Permutation-Based Proof

The complete proof for this subsection is avaliable in the file: PermProof.lean. As before, we prove that `insrt` operation preserves the permutation property.

```
theorem insrt_Perm (x : Nat) (l : List Nat) : List.Perm (x :: l) (insrt x l)
```

Respecting its inductive structure, the proof is quite simple. We proceed by induction on the input list, divide case by the location of the new element, and apply the inductive constructors for permutation.

- `l = nil`: Clear.
- `l = cons h t`: We have `List.Perm (x :: t) (insrt x t)` as inductive hypothesis. We perform case analysis on `x ≤ h`.
  - `x ≤ h`: Resulting list is `x :: h :: t`, which is same with `x :: l = x :: h :: t`.
  - `x > h`: Resulting list is `h :: insrt x t`. Since permutation is transitive, we can rewrite the goal as `h :: x :: t` using the inductive hypothesis. Then, we can achieve the goal by applying `Perm.swap`.

## 5.2 Multiset-Based Proof

The complete proof for this subsection is avaliable in the file: MultisetProof.lean.

```
theorem insrt_Contents (a: Nat)(l: List Nat): contents (insrt a l) = contents (a :: l)
```

The proof again proceeds by induction on the input list, together with case analysis. However, the multiset specification does not have any inductive constructors. Instead, we use the following algebraic property of multiset.

```
theorem union_swap (a b c : Multiset) : union a (union b c) = union b (union a c)
```

It states that swapping elements does not affect the contents. For detailed information, refer to the file: Spec.lean. Using this theorem, we can complete the proof of `insrt_Contents`.

- `l = nil`: Clear.
- `l = cons x xs`: We have `contents (insrt a xs) = contents (a :: xs)` as inductive hypothesis. We perform case analysis on `x ≤ h`.
  - `x ≤ h`: `contents (a :: x :: xs) = contents (a :: x :: xs)`.
  - `x > h`: Apply inductive hypothesis and `union_swap`.

## 5.3 Equivalence of the Two Approaches

Finally, we show that the permutation-based and multiset-based specifications are equivalent in the context of content preservation. The complete proof for this subsection is avaliable in the file: PermEq.lean.

```
theorem Perm_Contents (al bl: List Nat): List.Perm al bl→ contents al = contents bl
theorem Contents_Perm (al bl : List Nat): contents al = contents bl→ List.Perm al bl
```

**From permutation to multiset.** In this direction, we prove by induction on the premise, using the inductive definition of permutation.

- `Perm.nil`: Obvious.
- `Perm.cons x l`: By inductive hypothesis and the definition of `contents`.
- `Perm.swap x y l`: By the definition and `theorem union_swap`.
- `Perm.trans h1 h2`: By the inductive hypothesis.

**From multiset to permutation.** This direction requires much more work. We proceed by induction on the given lists, and then analyze cases by comparing their head elements. To deal with each case, we need several auxiliary lemmas including:

```
theorem Contents_nil_inv (l : List Nat) : (∀ x, 0 = contents l x) → l = []
theorem Contents_cons_inv (l : List Nat) (x n : Nat) :
  n + 1 = contents l x → ∃ l₁ l₂ , l = l₁ ++ x :: l₂ ∧ contents (l₁ ++ l₂) x = n
theorem Contents_insert_other (l₁ l₂ : List Nat) (x y : Nat) :
  y ≠ x → contents (l₁ ++ x :: l₂) y = contents (l₁ ++ l₂) y
```

Since this direction of the proof is considerably more intricate and is not essential for the main narrative, we do not include its full derivation here. For details, see PermEq.lean.

# 6 Discussion

For the sortedness property, our results suggest the following practical workflow when developing specifications in Lean: begin with an intuitive formulation that captures the informal idea, then develop a structurally convenient version that aligns better with Lean's inductive reasoning, prove their equivalence, and finally carry out the correctness proof using the convenient definition. This approach makes the specification both coherent and technically manageable.

For content preservation, the comparison between permutation and multiset formulations illustrates that the choice between multiple valid specifications can depend on the structural and algebraic properties. Each approach offers its own advantages, and users can select one based on the proof style they intend to adopt.

One interesting direction for future work concerns the definitions of `insrt` and `sort`. In this paper, we fixed them to the inductive, structurally recursive versions. However, one could instead define these functions numerically using list indices, which is standard in algorithm textbooks. Repeating the same comparative analysis with these alternative definitions may reveal additional relationships: for instance, numerical formulations might align more naturally with index-based specifications. Exploring how different combinations of algorithmic definitions and specification styles interact would deepen our understanding of proof design in Lean.

# 7 Conclusion

In this paper, we compared the proof of the insertion sort algorithm in Lean under different formulations of its correctness properties. For sortedness, we examined both an inductive formulation and an index-based formulation, and proved their equivalence. The inductive definition `Sorted` aligned well with the inductive nature of Lean proofs, whereas the numerical definition `Sorted'` was more intuitive, but resulted in a more technically involved proof, requiring careful reasoning about list indices and auxiliary lemmas.

For content preservation, we compared the permutation-based approach with a multiset-based approach. Both approach had their own advantages. Again, the inductive definition of permutation aligned well with the inductive nature of a Lean proof. On the other hand, the algebraic property of multiset allows concise reasoning.

The results of this study illustrate that correctness properties can often be specified in several valid ways, and that the choice of specification has a direct impact on the structure and complexity of the proofs. Although we cannot formally prove that our specification actually matches the informal, intended meaning, the equivalence between the specifications strengthens our confidence in the coherence of the definitions.

# References

[1] Andrew W. Appel. *Verified Functional Algorithms*, volume 3 of *Software Foundations*. Electronic textbook, 2025. Version 1.5.5, http://softwarefoundations.cis.upenn.edu.

[2] František SILVÁŠI and Martin TOMÁŠEK. Lean formalization of insertion sort stability and correctness. *Acta Electrotechnica et Informatica*, 18(2):42–49, Jun 2018.