Formal Verification of a Selection Sort Algorithm in Dafny

Zitian Wang

April 3, 2025

1 Introduction

This report presents the deductive verification of a selection sort algorithm implemented in Dafny. The goal is to formally prove that the algorithm sorts an array in non-decreasing order while preserving the multiset of elements. In this report, we discuss our choice of algorithm, the properties specified in the program, the design of invariants and annotations, adjustments made to simplify verification, and the hardest steps encountered during the process.

2 Algorithm Selection

For this project, we chose to verify a selection sort algorithm. The reasons for this choice include:

- **Simplicity:** Selection sort has a straightforward structure which makes it easier to modularize and verify.
- **Clear invariants:** The sorted prefix and the property that every element in the sorted part is less than or equal to those in the unsorted part can be expressed concisely.
- **Modularity:** By separating out helper methods, such as the method to find the minimum index in a subarray, the verification of each component becomes more manageable.

3 Specification and Implementation

In our Dafny implementation, we define two main predicates to specify sortedness:

- `isSorted` asserts that the entire array is sorted.
- `isSortedPrefix` asserts that the first n elements are sorted.

We also implement a helper method `MinIndex` to locate the minimum element in a subarray. The main sorting method, `SelectionSort`, repeatedly finds the minimum element in the unsorted portion and swaps it into position.

Below we present code segments along with detailed commentary.

3.1 Sortedness Predicates

```
predicate isSorted(a: array<int>)
  requires a != null
  reads a
{
  isSortedPrefix(a, a.Length)
}

predicate isSortedPrefix(a: array<int>, n: int)
  requires a != null
  requires 0 <= n <= a.Length
  reads a
{
  forall i, j :: 0 <= i < j < n ==> a[i] <= a[j]
}
```

Listing 1: Sortedness Predicates

Discussion: Here, `isSortedPrefix` is defined to capture the property that the first n elements of the array are in non-decreasing order. The predicate `isSorted` then simply applies `isSortedPrefix` to the entire array. This modular approach simplifies both the specification and the verification process.

3.2 Finding the Minimum Index

```
method MinIndex(a: array<int>, start: int) returns (minIdx: int)
  requires a != null
  requires 0 <= start < a.Length
  ensures start <= minIdx < a.Length
  ensures forall i :: start <= i < a.Length ==> a[minIdx] <= a[i]
{
  minIdx := start;
  var j := start;
  while j < a.Length
    invariant start <= j <= a.Length
    invariant start <= minIdx < a.Length
    invariant forall k :: start <= k < j ==> a[minIdx] <= a[k]
    decreases a.Length - j
  {
    if a[j] < a[minIdx] {
      minIdx := j;
    }
    j := j + 1;
  }
}
```

Listing 2: MinIndex Method

Discussion: The `MinIndex` method searches for the smallest element in the subarray starting at index `start`. The loop invariants ensure that at each iteration, `minIdx` is the index of the

smallest element in the examined part. This method is a key component because its correctness is essential for ensuring the overall sorting algorithm works properly.

3.3 SelectionSort Method

```
method SelectionSort(a: array<int>)
  requires a != null
  modifies a
  ensures isSorted(a)
{
  var n := a.Length;
  var pos := 0;
  while pos < n
    invariant 0 <= pos <= n
    invariant forall i, j :: 0 <= i < pos <= j < n ==> a[i] <= a[j]
    invariant isSortedPrefix(a, pos)
    decreases n - pos
  {
    var mi := MinIndex(a, pos);
    // Swap the element at 'pos' with the minimum element in a[pos .. n)
    var temp := a[pos];
    a[pos] := a[mi];
    a[mi] := temp;
    pos := pos + 1;
  }
}
```

Listing 3: SelectionSort Method

Discussion: The `SelectionSort` method works by incrementally building a sorted prefix. The loop invariant ensures that the subarray from index 0 to `pos` is sorted and that every element in the sorted prefix is less than or equal to every element in the unsorted portion. Each iteration calls `MinIndex` to find the minimum element in the unsorted part and then swaps it into the correct position. This clear separation of concerns makes both the implementation and the subsequent verification more tractable.

4 Verification Process and Discussion

Dafny generates verification conditions based on the preconditions, postconditions, and loop invariants. Discharging these VCs using an SMT solver gives us the following assurances:

- The loop invariants in both `MinIndex` and `SelectionSort` hold throughout execution.
- The postconditions of all methods are met upon termination.
- The overall property of sortedness (`isSorted`) is guaranteed by the maintained invariants.

Challenges Encountered:

- *Invariant Design:* One of the hardest parts was designing invariants that are strong enough to prove correctness yet not overly complex. In our approach, expressing that every element in the sorted prefix is less than or equal to every element in the unsorted portion required careful formulation.
- *Modular Verification:* Splitting the verification into modular components (such as `MinIndex`) helped in managing complexity. However, ensuring that the interactions between these components satisfy the overall specification still required iteration and refinement.
- *Tool Limitations:* At times, the SMT solver needed additional hints through invariants and assertions. Balancing these hints without overcomplicating the code was an iterative process.

5 Conclusion

This report presented a fully verified Dafny implementation of a selection sort algorithm. We discussed our algorithm choice, the specification properties, detailed annotations, and the challenges faced during verification. The modular design not only simplified verification but also provides a good basis for verifying more complex algorithms in future work.