Deductive Verification with Dafny: Software Analysis Assignment 1 Report

Airaghi Luca
Student ID: 24-995-813

April 3, 2025

1 Introduction

This report covers the implementation, formal specification, and verification of a sorting algorithm using the Dafny verification tool. The goal is to ensure the correctness of the chosen algorithm by proving that it meets its formal specification. The sorting algorithm selected for this assignment is *Selection Sort*.

2 Choice of Sorting Algorithm

For this assignment, I chose the *Selection Sort* algorithm because it is a simple and intuitive sorting algorithm. It works by repeatedly selecting the smallest element from the unsorted portion of the array and placing it in the correct position. This algorithm is useful for demonstrating basic verification techniques, as it is relatively easy to reason about. Before doing *Selection Sort* I did *QuickSort*, a more complex and interesting algorithm to analyze, but unfortunately I wasn't able to prove one of the conditions requested (sorted order of elements of the final array). For completeness, I leave the code of the QuickSort in GitLab.

2.1 Algorithm Overview

Selection Sort is a comparison-based in-place sorting algorithm that works by dividing the array into two sections: the left part is sorted, and the right part is unsorted. In each iteration, the algorithm finds the smallest element from the unsorted portion of the array and swaps it with the first element of the unsorted portion, thus expanding the sorted portion. This process is repeated until all the elements have been sorted.

In the first iteration, the algorithm scans all the elements in the array to find the smallest element. Then, place this smallest element at the beginning of the array. In subsequent iterations, the unsorted portion becomes smaller as the sorted portion grows. The algorithm continues until all elements are sorted.

3 Formal Specification

The formal specification of the sorting algorithm involves two main properties that must be proved:

- **Output is Sorted:** The output array must be sorted in increasing order.
- **Output is a permutation of input:** The output array must contain the same elements as the input array, just in a possibly different order.

These properties ensure that the sorting algorithm functions correctly by both ordering the elements and preserving their values.

3.1 Preconditions and Postconditions

The precondition for the algorithm is that the input array contains at least one element, and the postcondition guarantees that the output array is sorted and a permutation of the input.

4 Verification Process

In order to formally verify the correctness of the *Selection Sort* algorithm, several loop invariants were introduced. These invariants are critical in enabling Dafny to prove that the program maintains certain properties throughout its execution, such as preservation of array content, sortedness, and correctness of intermediate states.

4.1 Outer Loop Invariants

The outer loop iterates over the array, gradually expanding the sorted portion from left to right. To support verification, the following invariants were defined:

- **Bounds on** `i`: `0 <= i <= n - 1` and `0 <= i < arr.Length` ensure that the index `i` always stays within valid bounds during execution.
- **Sorted Prefix**:

  ```
  forall k, l :  int ::  0 <= k < l < i ==> arr[k] <= arr[l]
  ```

  This invariant guarantees that all elements before index `i` are sorted in non-decreasing order.

- **Prefix vs. Suffix Property**:

  ```
  forall k :  int :: 0 <= k < i ==>
                     forall j ::  i <= j < n ==> arr[k] <= arr[j]
  ```

  This invariant ensures that every element in the sorted portion of the array (from index `0` to `i-1`) is less than or equal to any element in the unsorted portion.

- **Permutation Preservation**:

  ```
  perm(arr[..], old(arr[..]))
  ```

  This ensures that the array remains a permutation of its original contents, confirming that no elements are lost or duplicated throughout sorting.

4.2 Inner Loop Invariants

The inner loop is responsible for scanning the unsorted portion of the array to find the smallest element. The invariants for this loop help prove that the correct minimum element is selected and that the array remains a permutation of the original:

- **Bounds on `j`**:

  ```
  i + 1 <= j <= n
  ```

  Ensures that the scanning index `j` remains within valid bounds of the unsorted portion.

- **Bounds on `min`**:

  ```
  i <= min < n
  ```

  Verifies that the index storing the current minimum candidate is always valid.

- **Minimum Element Tracking**:

  ```
  forall k ::  i <= k < j ==> arr[min] <= arr[k]
  ```

  This invariant is crucial to guarantee that `min` always points to the smallest element found between indices `i` and `j-1`. It allows Dafny to verify the correctness of the final swap operation after the inner loop finishes.

- **Permutation Preservation (Repeated)**:

  ```
  perm(arr[..], old(arr[..]))
  ```

  Reiterated in the inner loop to maintain the global property that the array is always a permutation of its original state.

4.3 Termination Proof

To prove termination, we defined a variant that measures the remaining work to be done by the algorithm. In this case, the variant was the number of elements left to be sorted. Dafny proves termination by inferring variants that strictly decrease with each loop iteration. For the outer loop, the variant is the remaining number of elements to sort, i.e., `n - i`. For the inner loop, the variant `n - j` decreases on each iteration. Since both `i` and `j` are incremented and bounded by `n`, the loops are guaranteed to terminate.

4.4 Challenges in Verification

The verification process was not without its challenges. One of the most complex aspects was handling the nested loops and ensuring that all loop invariants held throughout the execution. Specifically, proving that the array elements before the current index are always less than or equal to the remaining unsorted elements required careful attention to detail. The most difficult part however was to try to prove termination of *QuickSort* algorithm, which requires a deep understanding of how the algorithm works and how Dafny handles recursion.

5 Conclusion

In summary, the loop invariants used in both the outer and inner loops ensure that the array is sorted incrementally, the smallest elements are correctly identified, and the array contents remain unchanged aside from reordering. These properties, together with Dafny's support for verifying array permutations and ordering, enable full correctness verification of the algorithm.
This report detailed the implementation, specification, and verification of the *Selection Sort* algorithm using Dafny. The verification process was successful in proving the correctness of the algorithm with respect to the formal specification, and the challenges encountered were overcome with careful annotation and adjustment of the code.