FACULTY OF INFORMATICS

Verifying selection sort with Dafny

Software Analysis - Assignment 1

Authors FEDERICO
LAGRASTA

March 30, 2022

1 Selection sort

The sorting algorithm selected for verification is selection sort.
Selection sort can work both in place or build a new sorted array. The general idea of selection sort is to iteratively find the minimum in the unsorted part array and swap it with the first element of the unsorted array. At each iteration the sorted part of the array will increase its size by 1 until the algorithm terminates.
It's a very simple albeit inefficient algorithm with ($\theta(n^2)$) complexity.
Searching for the index of the smallest element of the unsorted part of the array can be written as an independent algorithm

```
method find_min_index(a : array<int>, s: int, e: int) returns (min_i:
int)
```

2 specification

Every sorting algorithm in place needs to satisfy three properties on the output for every legal input in order to be correct.

2.1 `out` is `in` permutation

The array in output must be a permutation of the original one (or the array passed as input must be a permutation of its old values if in place). This trivially because they must contain the same elements.
In order to represent this requirement I wrote the following function in Dafny

```
predicate is_permutation(a:seq<int>, b:seq<int>)
decreases |a|
decreases |b|
{
    |a| == |b|  &&
    ((|a| == 0 && |b| == 0) ||
    exists i,j : int :: 0<=i<|a| &&  0<=j<|b|  && a[i] ==
        b[j] && is_permutation(a[0..i] + if i < |a| then
       a[i+1..] else [], b[0..j] + if j < |b| then  b[j
       +1..] else []))
}
```

The idea was that in order for `a` to be a permutation of `b` they must have the same length and there must exist two indexes `i,j` of the arrays such that `a[i]==b[i]` **and** recursively the same must be true for the same arrays without the matching elements.
The function terminates because the the size of both arrays decreases with each call starting from a positive (obviously) value until it reaches 0 which is the base step of the recursion.
This should be an invariant of the loop but for some reason that I couldn't figure out it doesn't seem to work as such.
Apparently Dafny has support for multisets so the previous predicate can more easily be expressed as:

```
predicate is_permutation2(a:seq<int>, b:seq<int>)
{
    multiset(a) == multiset(b)
}
```

which does work.

2.2 The resulting array is sorted in ascending order

The elements of the output (or new values of the input if in place) must be sorted in ascending order.
This specification is expressed with the following predicate:

```
predicate is_sorted(ss: seq<int>)
{
    forall i, j: int:: 0 <= i <= j < |ss| ==> ss[i] <= ss
       [j]
}
```

For every possible pair of indexes the element indexed by the smaller or equal one must be smaller or equal.

3 The find min index method

As mentioned before the following method finds the index indexing one of the smallest elements in a portion of the array `[s,e)` with the following:

```
ensures forall k: int :: s <= k < e ==> a[min_i] <= a[k]
```

The method **requires** `s,e` to be valid indexes of the array which has size of at least 1 and additionally for `e` to be strictly greater than `s`. In this manner the verification of the sorting algorithm will always return a valid index and won't have to deal with values such as `-1` or `null`.
The output of the algorithm is ensured to be a valid index of the array in input such that the element indexed by it is at most equal to the other elements of the array.
Given that `e` is strictly larger than `s` we don't need to require `e >= 0`.
The loop variant in this case is `V = e - i`.
As invariants we require `i` to be in [`s`, `e`] and `min_i` to always be a real index of the array.
To then prove the validity of the requirement:

```
ensures forall k: int :: s <= k < e ==> a[min_i] <= a[k]
```

we use it as an invariant with `e` replaced by `i` since at the end of the loop we will have `i >= e` which will prove the requirement combined with the other invariants.
The invariant

```
invariant i < e ==> min_i <= i
```

is unnecessary since already implied by:

```
invariant s <= i <= e
invariant s <= min_i < e
```

```
method find_min_index(a : array<int>, s: int, e: int)
   returns (min_i: int)
requires a.Length > 0
requires 0 <= s < a.Length
requires e <= a.Length
requires e > s

ensures min_i >= s
ensures min_i < e
ensures forall k: int :: s <= k < e ==> a[min_i] <= a[k]
{
    min_i := s;
    var i : int := s;

    while i < e
    decreases e - i // loop variant
    invariant s <= i <= e
    invariant s <= min_i < e
    // unnecessary invariant
    // invariant i < e ==> min_i <= i
    invariant forall k: int :: s <= k < i ==> a[min_i] <=
        a[k]
    {
        if a[i] <= a[min_i] {
            min_i := i;
        }
        i := i + 1;
    }

}
```

4 The selection sort method

The method originally required
`ns.Length >= 0`
which is obviously not needed.
Another requirement would be:
`require ns != null`
but this is implicit in the `array<int>` type which does not allow null (conversely to `array?<int>`).
As mentioned previously we need to ensure the output is a permutation of the input and the output is sorted.
We also need to declare that we are potentially modifying the input array.
As invariant for the permutation requirement we can directly use the `is_permutation2(old(ns[..]), ns[..])` predicate with the new values assigned to the array.
For the array being sorted we instead use as invariant the fact the the elements to

the left of the loop variable `i` must already be sorted so at the end of the loop every value in the array will be sorted since we will have `i == ns.Length` for the other invariants.

```
method selection_sort(ns: array<int>)
// requires ns.Length >= 0
ensures is_sorted(ns[..])
ensures is_permutation2(old(ns[..]), ns[..])
modifies ns
{
    var i: int := 0;
    var l: int := ns.Length;
    while i < l
    decreases l - i
    invariant 0 <= i <= l
    invariant is_permutation2(old(ns[..]), ns[..])
    invariant forall k, kk: int :: 0 <= k < i  && i <= kk
        < ns.Length ==> ns[k] <= ns[kk] // left els must
       be lesser than right ones
    invariant is_sorted(ns[..i])
    {
        var min_i: int := find_min_index(ns, i, ns.Length
           );
        ns[i], ns[min_i] := ns[min_i], ns[i];
        i := i + 1;
    }

}
```