Assignment 3 - Insertion Sort Using Dafny

Gaurang Tandon

April 27, 2020

Contents

1 Insertion sort definition

1. In insertion sort, the base case is that the subarrray of size 1 in front of the array is considered sorted by default.

2. Now, assuming subarray $(1, i)$ is sorted, we take the (i+1)-th element and insert it into its correct position in the sorted subarray $(1, i)$. Either it will remain at its position or get inserted somewhere in the middle.

3. Step 2 will result in subarray $(1, i + 1)$ being sorted.

4. After having proved base case and inductive step, we can now say that eventually insertion sort will sort the whole array, the subarray $(1, arr.Length)$

5. Regarding its termination, we note that step 2 cannot take more than n steps, and if we repeat that step n times, we have a complexity of at most $\mathcal{O}(n^2)$, therefore we are guaranteed that the program terminates.

6. We shall attempt to prove this formally in below.

2 Transition System Definition

1. $S_{ins} =< X, X^o, U, \rightarrow, Y, h >$

2. The state space of the system $X = \mathbb{Z}^{\mathbb{N}} \times \mathbb{N}$ (integer array of natural length)

3. We define a function $\rho : \mathbb{N} \rightarrow X$, which converts the input space of the problem to the state space of the system

4. $\rho(n) = (\text{arr}, 1)$ is the case for the initial state. Hence, $X^o = \rho(n) = (\text{arr}, 1)$.

5. U = {next}

6. Transition Relation:

   - $(\text{arr}, i) \xrightarrow[\text{next}]{\text{sort-pass}} (\text{arr'}, i + 1)$, such that the i+1-th element gets sorted in this pass. By "element gets sorted", we mean that the i+1-th element gets inserted in its correct (sorted) position in the subarray from first position to i+1-th position.

- (arr, n) is terminal state where $n = \text{arr.Length}$

7. We define a transition function $t : X \to X$, and $\mathrm{t^n}$ as the $\mathrm{n^{th}}$ iteration of function t, where $n \in \mathbb{Z} \wedge n > 0$ defined by $t^0 = t, t^1 = t \circ t, t^n = t \circ t....(n-1)times.... \circ t = t \circ t^{n-1}$

8. Let $\mathrm{X_f}$ be the final state of the system, defined as $X_f = t^n(\text{arr}, 1)$ such that `arr` is sorted. Now $\mathrm{t^0}$ corresponds to $\mathrm{X^o}$, and likewise $\mathrm{t^n}$ corresponds to $\mathrm{X_f}$. Which means $X^o \xrightarrow{*} X_f = t^n$

9. $Y = \mathbb{Z}^{\mathbb{N}}$, as the view space of the system is equal to the output space of the problem.

10. $h : X \to Y$, where $h : X \to \mathbb{Z}^{\mathbb{N}}$

3 Insertion sort Program

```
datatype StateSpace = StateSpace(arr: array<int>, pass: int)

predicate ordered(x: int, y: int) {
x <= y
}

predicate sorted(arr: array<int>, start: nat, end: nat)
reads arr
requires 0 <= start <= end < arr.Length
{
forall x :: start < x <= end ==> ordered(arr[x - 1], arr[x])
}

method InsertionSortStateTransition(state: StateSpace) returns (finalState: StateSpace
modifies state.arr
requires state.arr.Length >= 1
requires state.pass == 1
ensures state.arr.Length == finalState.arr.Length
ensures sorted(finalState.arr, 0, finalState.arr.Length - 1)
ensures finalState.pass == finalState.arr.Length
{
var arr := state.arr;
var n := arr.Length;
var i := 1;
```

```
while i < n
invariant i <= arr.Length;
invariant sorted(arr, 0, i - 1)
modifies arr {
var j := i;

while j >= 1 && arr[j] < arr[j - 1]
// ensure that left and right half of
// the arrays are themselves sorted
invariant j >= 1 ==> sorted(arr, 0, j - 1)
invariant j < i ==> sorted(arr, j + 1, i)

// now that our array is split into three parts
// values to left of j, value at j, values to right of j
// establish ordering relations between them
invariant j < i ==> ordered(arr[j], arr[j + 1])
invariant 1 <= j < i ==> ordered(arr[j - 1], arr[j + 1])
modifies arr {
arr[j], arr[j - 1] := arr[j - 1], arr[j];
j := j - 1;
}

i := i + 1;
}

finalState := StateSpace(arr, arr.Length);
}

function method rho(arr: array<int>) : StateSpace {
StateSpace(arr, 1)
}

function method pi(st: StateSpace) : array<int> {
st.arr
}

method Main(){
var arr := new int[5];
arr[0], arr[1], arr[2], arr[3], arr[4] := 2, 1, 3, 4, 5;
```

```
var sts := rho(arr);
var sts2 := InsertionSortStateTransition(sts);
var sortedArr := pi(sts2);

var i := 0;

while i < sortedArr.Length {
print sortedArr[i];
i := i + 1;
}

assert sorted(sortedArr, 0, sortedArr.Length - 1);
}
```

4 Conditions

4.1 Pre conditions (`requires`)

- arr.Length ≥ 1

4.2 Post conditions (`ensures`)

- initialState.arr.Length $==$ finalState.arr.Length
- finalState.arr $=$ sorted result of initialState.arr

5 Justification of Hoare logic

5.1 Definitions

5.1.1 ordered(x, y)

`ordered` is a comparator over two elements x and y, and it is true iff both the following are true:

1. they are comparable with this comparator
2. this comparator would place x before y

5.1.2 sorted(sequence)

A sequence `arr` (of length n) is sorted iff one of the following is true:

1. it is empty
2. it has one element
3. $\forall i$ such that $0 \leq i \leq n-1$, `ordered(arr[i], arr[i + 1])` holds for some transitive comparator `ordered`, as defined in 5.1.1.

5.2 Lemmas

5.2.1 Subarray sorted

If a sequence of elements $arr = (a_0, a_1, \ldots, a_n)$ is sorted, such that $n \geq 1$, then the sequence $(a_0, \ldots, a_{n-1})$ is also sorted with the same comparator.

Proof:

1. $\forall i$ such that $0 \leq i \leq n-1$, `ordered(arr[i], arr[i + 1])` holds for some transitive comparator `ordered`, because:
   - 5.1.2 (point 3)
2. $\forall i$ such that $0 \leq i \leq n-2$, `ordered(arr[i], arr[i + 1])` holds for same transitive comparator, because:
   - point 1 above
3. sequence of elements $(a_0, \ldots a_{n-1})$ is sorted, because
   - 5.1.2
   - point 2 above

5.2.2 Larger subarray sorted

If a sequence of elements $arr = (a_0, a_1, \ldots, a_n)$ is sorted, such that $n \geq 0$, then the sequence $(x, a_0, \ldots, a_n)$ is also sorted with the same comparator, given `ordered(x, arr[0])` for any element x.

Proof:

1. $\forall i$ such that $0 \leq i \leq n-1$, `ordered(arr[i], arr[i + 1])` holds for some transitive comparator `ordered`, because:

- 5.1.2

2. sequence of elements $(x, a_0, \dots a_n)$ is sorted, because

   - 5.1.2
   - given `ordered(x, arr[0])` holds

5.3 Notation

Notation is same as in Hoare Logic Rules PDF shared on Microsoft Teams. We define:

1. P as the precondition
2. S as a single statement
3. Q as the loop invariant
4. n is the length of the array.

5.4 Inner loop

5.4.1 Description

Given an index j, one iteration of the inner loop moves the element at index j one step left, as long as remains unsorted. Described in detail in 1.

5.4.2 Partial correctness

We define the following terms:

1. `left array`: subarray to the left of `j`-th element, i.e., `arr[0..j-1]`, denoted as $a_0, \dots a_p$
2. `right array`: subarray to the right of `j`-th element, i.e., `arr[j+1..n-1]`, denoted as $b_0, \dots b_q$
3. P: $j \geq 1$ and the index j has an inversion (i.e., $\mathrm{arr[j]} \leq \mathrm{a_p}$)
4. S: swap values in array at index j and $j-1$, and decrement j by one
5. Q: the logical and of four conditions:

   (a) Q_1 - the left array is sorted

(b) Q_2 - the right array is sorted

(c) Q_3 - ordered(a_p, b_0)

(d) Q_4 - ordered(arr[j], b_0)

We need to show that after one execution of S, the invariant continues to hold.

Let us execute one step of S. Now, the left array is $a_0, \dots a_{p-1}$ and right array is $a_p, b_0 \dots b_q$. And j has decremented by 1.

Now, we check all four invariant conditions:

1. Q_1' holds because $Q_1 \Rightarrow$ Lemma 5.2.1
2. Q_2' holds, by combining Q_3 and Q_2 with Lemma 5.2.2
3. Q_3' holds, since ordered($\mathrm{a_{p\text{-}1}}$, $\mathrm{a_p}$) ($\because$ sequence a is sorted)
4. Q_4' holds, since P

Therefore, since we have proved that {P $\wedge$ Q}S{Q} holds, hence {Q}(WHILE P DO S){$\mathrm{P^c}$ $\wedge$ Q} holds by using the **while rule for partial correctness**

5.4.3 Total correctness

To prove: total correctness ([P $\wedge$ Q]), i.e., the condition does not remain true forever if we execute statement S.

Proof by contradiction

Assumption: P remains true forever.

Proof:

1. S will execute forever, because:
   - assumption
   - Q remains true since it is loop invariant
   - above two points imply P $\wedge$ Q always true, which is the entry condition for S
2. j will decrease forever, because:
   - S will execute forever, by point 1 above
   - every execution of S decreases j by one

3. $j \geq 1$ holds forever, because:

    - splitting up assumption

Contradiction: both point 2 and 3 cannot hold since j is a finite integer.

Conclusion: false assumption, P does not remain true forever. Hence, we conclude that [Q](WHILE P DO S)[P^c $\wedge$ Q] holds considering [P $\wedge$ Q]S[Q], given by **while rule for total correctness**.

5.5 Outer loop

5.5.1 Description

One iteration of the outer loop increases the size of the sorted subaray, from $(0, p)$ to $(0, p+1)$, where p is the pass number.

5.5.2 Partial correctnesss

We define the following terms:

1. P: $i < n$

2. S: execute the inner loop, setting $j = i$, and then increment i by one

3. Q: the logical and of two conditions:

    (a) Q_1 - $i \leq n$

    (b) Q_2 - the sequence `(arr[0], ..., arr[i - 1])` is sorted

We need to show that after one execution of S, the invariant continues to hold.

Let us execute one step of S.

Now, we check the two invariant conditions:

1. Q_1' holds, since $i < n$ before execution of S (guaranteed by P), and one step of S increases i by only one.

2. Q_2' holds, since even when i is increased by one, execution of S guarantees that the subarray $(0, i)$ remains sorted (proved in earlier section).

Therefore, since we have proved that {P $\wedge$ Q}S{Q} holds, hence {Q}(WHILE P DO S){P^c $\wedge$ Q} holds by using the **while rule for partial correctness**

5.5.3 Total correctness

To prove: total correctness ([P $\wedge$ Q]), i.e., the condition does not remain true forever if we execute statement S.

Proof by contradiction

Assumption: P remains true forever.

Proof:

1. S will execute forever, because:
   - assumption
   - Q remains true since it is loop invariant
   - above two points imply P $\wedge$ Q always true, which is the entry condition for S
2. i will increase forever, because:
   - S will execute forever, by point 1 above
   - every execution of S increases i by one
3. $i < n$ holds forever, because:
   - splitting up assumption

Contradiction: both point 2 and 3 cannot hold since n is a finite integer.

Conclusion: false assumption, P does not remain true forever. Hence, we conclude that [Q](WHILE P DO S)[P^c $\wedge$ Q] holds considering [P $\wedge$ Q]S[Q], given by **while rule for total correctness**.