Software Testing, Quality Assurance and Maintenance Winter 2026

Lecture 10 — February 6, 2026

Patrick Lam *version 1*

We are now going to verify a bubble sort implementation, which will be helpful for doing the assignment. I am assuming that everyone here has seen bubble sort at some point in the past. It's been longer for me than it's been for you. (Though I wouldn't expect bubble sort to be in CS 341). And, spoiler, there is going to be a twist in the verification.

You can find the bubble sort implementation at bubble_sort.prob.dfy in the Dafny repo.

Auxiliary predicates: sorted and pivot. First, we are going to define a predicate sorted. We've done similar things before. This time we specify a from and to range to check for sortedness. The predicate returns true iff its array parameter a is sorted between the range [from, to).

```
predicate sorted(a: array<int>, from: int, to: int)
  requires 0 ≤ from ∧ to ≤ a.Length
  reads a
{
  ∀ j, k • from ≤ j < k < to ⟹ a[j] ≤ a[k]
}
```

Next, I'm going to introduce a pivot predicate. I'm going to do this before a reminder of how bubble sort works. You may remember pivots being useful, in general, for sorting algorithms.

pivot takes three arguments: an array, a to argument, and a potential pvt. This predicate checks the sub-array [0, to) of its array for whether pvt is a pivot—that is, all elements with index less than pvt are less than (or equal to) all elements of the sub-array with index greater than pvt.

We can encode the predicate as follows:

```
predicate pivot(a: array<int>, to: int, pvt: int)
  requires 0 ≤ pvt ≤ to ≤ a.Length
  reads a
{
  ∀ u, v • 0 ≤ u < pvt < v < to ⟹ a[u] ≤ a[v]
}
```

So, we let u take all values in $[0, \mathsf{pvt})$; and v is in $[\mathsf{pvt}+1, \mathsf{to})$. The predicate is true iff all a[u] are less than or equal to all a[v].

Dafny doesn't complain about this definition, but how do we know that the words correspond to the formalism?

We don't.

We can look at the formalism closely. It seems to be OK, but that is not a guarantee.

Another thing we can do is to run some test cases. I've included both assert and expect checks. In this case, in the Dafny model, probably assert is better: it is possible to verify these statically. For a method, all we check is the postcondition. But this is a function and so we are running the body when we use assert.

```
method {:test} TestPivot()
{
  var singleton := new int[] [4];
  assert pivot(singleton, singleton.Length, 0) = true;
  var three := new int[] [2,3,4];
  expect pivot(three, three.Length, 1) = true;
  var threeB := new int[] [4,3,2];
  expect pivot(threeB, threeB.Length, 1) = false;
  var a := new int[] [3, 5, 9, 2, 11, 13, 15, 12];
  expect pivot(a, a.Length, 0) = true;
  // a[3] < a[1], so false:
  expect pivot(a, a.Length, 2) = false;
  expect pivot(a, a.Length, 3) = true;
}
```

Dafny statically verifies the assert and, when you run the test case, it passes.

Implementing bubble sort. Here is an implementation of bubble sort. Note that it modifies array a in place.

```
method bubbleSort (a: array<int>)
{
  var i:nat := 1;

  while (i < a.Length)
  {
    var j:nat := i;
    while (j > 0)
    {
      if (a[j-1] > a[j]) {
        // Swap a[j-1] and a[j]
        var temp := a[j-1];
        a[j] := temp;
        a[j-1] := a[j];
      }
      j := j - 1;
    }
    i := i+1;
  }
}
```

Let's now review a bit how this works. We know that bubble sort is $O(n^2)$. The outer loop ensures that the subarray from $[0, i)$ is sorted at each iteration. The inner loop makes this happen by bubbling down items inside $[0, i)$ that are out-of-order, starting at index i and pushing them as far as needed, possibly to index 0.

For instance:

```
9, 3, 5, 2
3, 9, 5, 2
3, 5, 9, 2
3, 5, 2, 9
3, 2, 5, 9
2, 3, 5, 9
```

In the lecture notes I'm not showing the loop iterations explicitly, but it's a worthwhile exercise to write the array contents at each stage, and in particular, to note which sub-arrays are sorted.

Let's write a contract. This is perhaps an obvious contract for this method.

```
  requires a.Length > 0
  ensures sorted(a, 0, a.Length)
  modifies a
```

And we know that we have to write invariants for the `while` loop. We've done this several times already, so we know that this invariant is likely to work:

```
  invariant i ≤ a.Length
  invariant sorted(a, 0, i)
```

As we might expect, because we haven't provided any invariants at all for the inner loop, all Dafny knows coming out of the inner loop is that $j = 0$, which of course does not imply that any sub-array is sorted.

What I did next was look at the expanded loop iterations to see which sub-arrays were sorted. One pattern that is possible to see is that, of course, the sub-array $[0, i)$ always remains sorted in the inner loop; and also, sub-array $[j, i + 1)$ also remains sorted. So we can write this inner loop invariant:

```
  invariant 0 ≤ j ≤ i ≤ a.Length
  invariant sorted(a, 0, i)
  invariant sorted(a, j, i+1)
```

and Dafny doesn't complain.

It might seem like we're done. That was easy?

Are you sure? Let's write a test case.

```
method {:test} TestSort()
{
  var a := new int[] [3, 5, 9, 2, 11, 13, 15, 12];
  bubbleSort(a);
  var b := new int[] [2, 3, 5, 9, 11, 12, 13, 15];
  expect a[..] = b[..];
}
```

Dafny's arrays, like many other languages, are object-like. It's like Java. Comparing == of two arrays is probably not what you want, and definitely not here. We want something like Java's `equals()`, where we check the contents of the arrays. To do that in Dafny, we can form sequences (a Dafny built-in type) from the arrays, and compare those.

If we run this test case, Dafny reports that it fails. (Since we are calling a method, this really has to be `expect` and not `assert`). Why is this? The test harness output is not so helpful, but we can add `print a[..]` to the test case, printing out the sequence:

```
Dafny program verifier finished with 10 verified, 0 errors
TestPivot: PASSED
TestSort: [3, 5, 9, 9, 11, 13, 15, 15]FAILED
  bubble_sort.prob.dfy(88,1): expectation violation
TestSort2: PASSED
[Program halted] bubble_sort.prob.dfy(12,0): Test failures occurred: see above.
```

Uh oh. That's not what we expected. Somehow there are two 9s and two 15s in the array.

Multisets to the rescue. We have seen that the output is wrong. Even though it verifies. The array is indeed sorted, but it's not at all equal to a permutation of the original array. Our ensures clause is too weak, and forgot to specify that part of the relationship between the old array and the new array. Indeed, a function could just return the empty array and still satisfy the specification.

Permutations are actually not so obvious in Dafny. But Dafny has primitive type multiset, which stores elements and their multiplicities. If the output array is sorted, and if converting both arrays to multisets and comparing them succeeds, then we can say that we have a sort implementation. Let's do that.

```
method bubbleSort2 (a: array<int>)
  requires a.Length > 0
  ensures sorted(a, 0, a.Length)
  ensures multiset(a[..]) = multiset(old(a[..]))
  modifies a
```

Here, old refers to the contents of a upon entry to the method. Dafny has to be able to reason about those while verifying, so that we can compare old and new.

Of course, bubbleSort2 doesn't verify. It shouldn't! We add the multiset constraint as an invariant to both loops as well. Now, Dafny says that the multiset invariant is not preserved by the inner loop. That's odd... the contents are changing in the inner loop? Oh! It turns out that someone put wrong code to swap array elements. There was a bug. Despite verification (of the wrong contract). Let's fix that.

```
  var temp := a[j-1];
  a[j-1] := a[j];
  a[j] := temp;
```

Now Dafny doesn't complain about the multiset invariant anymore. It does complain about how sortedness from 0 to i is not maintained by the loop.

I did mention the pivot predicate a while ago. And the first sorted call from 0 to i seems somewhat fishy. It would make more sense to have that call range from 0 to j, which is the loop variable here. And it would also make sense for a[j] to be the pivot of the sub-array between 0 and i + 1. So, this invariant actually works for the inner loop:

```
        invariant 0 ≤ j ≤ i ≤ a.Length
        invariant sorted(a, 0, j)
        invariant pivot(a, i+1, j)
        invariant sorted(a, j, i+1)
        invariant multiset(a[..]) = multiset(old(a[..]))
```

That's all the insight I can give you about where the corrected invariants come from.

Dafny does verify this! It would be smart to still be a bit suspicious. But we can cross-check the verification with a test case.

```
method {:test} TestSort2()
{
  var a := new int[] [3, 5, 9, 2, 11, 13, 15, 12];
  bubbleSort2(a);
  var b := new int[] [2, 3, 5, 9, 11, 12, 13, 15];
  expect multiset(a[..]) = multiset(b[..]);
}
```

which passes. Between the test case (which is not super strong evidence) and the verification, we can be more sure that this implementation is correct, though again, there are still no guarantees.