Instructor: Matt Fredrikson TA: Ryan Wagner

Incremental Proof Development in Dafny

In this discussion, we'll see in more detail how to go about proving the total correctness of imperative code in Dafny. We'll use the precondition method discussed in last lecture to develop loop invariants and assertions that are strong enough to establish our specification, while simultaneously developing the proof in Dafny.

The program that we'll consider implements bubble sort, a basic sorting algorithm with worst-case performance $O(n^2)$, where n is the number of elements to be sorted. Although there are more efficient sorting algorithms, bubble sort is a good way of illustrating several techniques and concepts that are useful when reasoning about the correctness of imperative programs.

Partial Correctness

We begin by writing the basic code, absent any specification or annotation. Recall that bubble sort takes a single array as input, and is composed of a nested loop. The outer loop begins indexing at the end of the array, working its way towards the beginning, while the inner loop starts at the beginning of the array and works towards the current index of the outer loop. With each iteration of the outer loop, the largest element from the beginning of the array is moved towards the end by swapping out-of-order pairs in the inner loop. In this way, the loop becomes more sorted with each iteration of the outer loop by ensuring that elements in the range $[\texttt{i}, \texttt{a.Length} - 1]$ are always sorted.

```
method BubbleSort(a: array<int>)
{
  var i := a.Length - 1;
  while(i > 0)
  {
    var j := 0;
    while(j < i)
    {
      if(a[j] > a[j+1])
      {
        a[j], a[j+1] := a[j+1], a[j];
      }
      j := j + 1;
    }
    i := i - 1;
  }
}
```

Regarding the specification, our basic requirement is that whenever `BubbleSort` terminates, the contents of `a` are in sorted ascending order. We encode this requirement in a predicate. The predicate is *true* whenever for any indices $\texttt{i} \leq \texttt{j}$ within the index bounds, $\texttt{a}[\texttt{i}] \leq \texttt{a}[\texttt{j}]$.

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

`sorted` takes three arguments: the array `a`, a lower-bound index `l`, and an upper-bound index `u`. Note that our predicate essentially ignores calls with invalid index bounds, so that if $0 \leq \texttt{l} \leq \texttt{u} < \texttt{a.Length}$ is *false*, then the predicate will always return *true* regardless of the contents of `a`. There are two additional annotations present in the signature of `sorted`:

1. `reads a` tells Dafny that this predicate will read the contents of `a`. Whenever we write a function or predicate that examines the contents of an array, we need to use this annotation to let Dafny know.

2. `requires a != null` is a precondition on our predicate which says that we are not allowed to invoke `sorted` on an array reference that might be set to `null`. This means that we are not defining sortedness on `null` array references, so whenever we want to reason about this predicate, we must prove that the reference we pass takes a proper value.

Having defined what it means for an array to be sorted in ascending order, we can write the postcondition for our specification to state that `sorted` holds for every index in `a` (i.e., between 0 and $\texttt{a.Length} - 1$):

```
  ensures sorted(0, a, a.Length-1)
```

Because our postcondition stipulates that the reference we're passing to `sorted` isn't `null`, we will also need this precondition in our method. This is also evident by the lack of dynamic checks in `BubbleSort` for `null` values; if we didn't have this precondition, we could potentially dereference a `null` pointer:

```
  requires a != null
```

Looking for basic facts to add to our annotations, we start by observing that the outer loop index variable `i` will always lie in the range $[-1, \texttt{a.Length} - 1]$. For the inner loop, we can write a stronger invariant for `i`, namely that it will lie in the range $[1, \texttt{a.Length} - 1]$. Additionally, the inner-loop index variable `j` will lie in the range $[0, i]$, giving us our initial set of basic annotations.

```
method BubbleSort(a: array<int>)
  requires a != null
  ensures sorted(a, 0, a.Length-1)
{
  var i := a.Length - 1;
  while(i > 0)
    invariant -1 <= i < a.Length
  {
    var j := 0;
    while(j < i)
      invariant 0 < i < a.Length && 0 <= j <= i
    {
      if(a[j] > a[j+1])
      {
        a[j], a[j+1] := a[j+1], a[j];
      }
      j := j + 1;
    }
    i := i - 1;
  }
}
```

These annotations are clearly not strong enough to prove our desired postcondition, as they say nothing about the contents of `a`. We'll need to strengthen them, and in particular the loop invariants. Starting with the outer loop, we propagate the postcondition backwards to the point where the outer loop terminates. This will tell us what conditions need to hold when the loop exits to support the postcondition, which will hopefully guide our intuition towards better loop invariants.

```
invariant -1 <= i < a.Length
assume i <= 0
ensures sorted(a, 0, a.Length-1)
```

Computing the verification condition,

$$\begin{aligned} & -1 \leq \texttt{i} < \texttt{a.Length} \rightarrow \mathsf{wlp}(\texttt{assume i} \leq 0, \texttt{sorted}(\texttt{a},\ 0,\ \texttt{a.Length} - 1)) \\ \Leftrightarrow\ & (-1 \leq \texttt{i} < \texttt{a.Length} \wedge \texttt{i} \leq 0) \rightarrow \texttt{sorted}(\texttt{a},\ 0,\ \texttt{a.Length} - 1) \\ \Leftrightarrow\ & -1 \leq \texttt{i} \leq 0 \rightarrow \texttt{sorted}(\texttt{a},\ 0,\ \texttt{a.Length} - 1) \end{aligned}$$

This isn't all that surprising, and just points out that when the loop terminates, if `i` is either -1 or 0, then the contents of `a` should be sorted. Informally, we know that `i` will be either -1 or 0 when the loop terminates, and if $\texttt{i} = -1$, then `sorted` will necessarily be true, so this fact doesn't seem to tell us anything new. We want to generalize this fact to something that reflects the progress made at each iteration of the outer loop, such that when it is conjoined with the negated loop guard $\neg(\texttt{i} > 0)$, the postcondition will hold. After a bit of thought, we arrive at:

$$\texttt{sorted}(\texttt{a},\ \texttt{i},\ \texttt{a.Length} - 1)$$

Adding this to our outer loop invariant makes sense, as it corresponds to our observation that as the method proceeds, it maintains the fact that the array remains sorted in the range $[\texttt{i}, \texttt{a.Length} - 1]$. Additionally, we see that:

$$\neg(\texttt{i} > 0) \wedge \texttt{sorted}(\texttt{a},\ \texttt{i},\ \texttt{a.Length} - 1) \rightarrow \texttt{sorted}(\texttt{a},\ 0,\ \texttt{a.Length} - 1)$$

Continuing on, we need to find an invariant for the inner loop that will be sufficient to preserve our outer invariant. Applying the precondition method from the exit point of the inner loop to the next iteration of the outer loop, we have:

```
invariant 0 < i < a.Length && 0 <= j <= i
assume i <= j;
i := i - 1
invariant -1 <= i < a.Length && sorted(a, i, a.Length-1)
```

We can effectively ignore the portion of the postcondition constraining the bounds of `i`, as a straightforward calculation will verify that they are preserved by the loop. Then, computing the corresponding verification condition:

$$
\begin{array}{ll}
 & 0 < \texttt{i} < \texttt{a.Length} \wedge 0 \leq \texttt{j} \leq \texttt{i} \rightarrow \mathsf{wlp}(\texttt{assume i} \leq \texttt{j}; \texttt{i} := \texttt{i} - 1, \texttt{sorted(a, i, a.Length} - 1)) \\
\Leftrightarrow & (0 < \texttt{i} < \texttt{a.Length} \wedge 0 \leq \texttt{j} \leq \texttt{i} \wedge \texttt{i} \leq \texttt{j}) \rightarrow \texttt{sorted(a, i} - 1, \texttt{a.Length} - 1) \\
\Leftrightarrow & (0 < \texttt{i} < \texttt{a.Length} \wedge 0 \leq \texttt{j} \wedge \texttt{i} = \texttt{j}) \rightarrow \texttt{sorted(a, i} - 1, \texttt{a.Length} - 1)
\end{array}
$$

We might be tempted to generalize this fact by taking the consequent and adding it to our inner loop invariant, giving us:

```
method BubbleSort(a: array<int>)
  modifies a
  requires a != null
  ensures sorted(a, 0, a.Length-1)
{
  var i := a.Length - 1;
  while(i > 0)
    invariant -1 <= i < a.Length
    invariant sorted(a, i, a.Length-1)
  {
    var j := 0;
    while(j < i)
      invariant 0 < i < a.Length && 0 <= j <= i
      invariant sorted(a, i-1, a.Length-1)
    {
      if(a[j] > a[j+1])
      {
        a[j], a[j+1] := a[j+1], a[j];
      }
      j := j + 1;
    }
    i := i - 1;
  }
}
```

This would make the basic path from the beginning of the outer loop to the beginning of the inner loop:

```
invariant -1 <= i < a.Length && sorted(a, i, a.Length-1)
assume 0 < i;
j := 0
invariant 0 < i < a.Length && 0 <= j <= i && sorted(a, i-1, a.Length-1)
```

Which would lead to the verification condition:

$$0 < \texttt{i} < \texttt{a.Length} \wedge \texttt{sorted}(\texttt{a},\ \texttt{i},\ \texttt{a.Length} - 1) \rightarrow \texttt{sorted}(\texttt{a},\ \texttt{i} - 1,\ \texttt{a.Length} - 1)$$

This is not valid, as there is nothing in the antecedent to imply that a larger portion of the list is sorted (i.e., $[\texttt{i} - 1, \texttt{a.Length} - 1]$) than what was assumed (i.e., $[\texttt{i}, \texttt{a.Length} - 1]$). We can observe the following counterexample as concrete evidence that this condition is invalid:

1	2	3	5	4
			(j)	*(i)*

This array satisfies the antecedent, because it is sorted starting at `i`. It does not satisfy the consequent, because it is not sorted starting at $\texttt{i} - 1$. To find a suitable invariant, we will need to give this more thought. In the meantime, we will add the main invariant on the outer loop,

$$\texttt{sorted}(\texttt{a},\ \texttt{i},\ \texttt{a.Length} - 1)$$

to the inner loop, as we know that at the very least, it must be preserved.

It oftentimes helps to step through the execution of the code in question, examining they key invariants that arise in actual traces. Let's look at a small trace of bubble sort on the array $[4, 1, 3, 2]$.

4	1	3	2	*Swap 4, 1*
(j)			*(i)*	
1	4	3	2	*Swap 4, 3*
	(j)		*(i)*	
1	3	4	2	*Swap 4, 2*
		(j)	*(i)*	
1	3	2	4	
			(j) (i)	
1	3	2	4	
(j)		*(i)*		
1	3	2	4	*Swap 3, 2*
	(j)	*(i)*		
1	2	3	4	
		(j) (i)		
1	2	3	4	
(j)	*(i)*			
1	2	3	4	
	(j) (i)			
1	2	3	4	
(j) (i)				

Considering the outer invariant we're currently trying to support, one fact stands out immediately: throughout execution, all elements of `a` indexed by a value greater than `i` are greater than those that are at most `i`. Formally, we observe the invariant:

$$\forall \texttt{k}, \texttt{k}'.0 \leq \texttt{k} \leq \texttt{i} \wedge \texttt{i} < \texttt{k}' < \texttt{a.Length} \rightarrow \texttt{a}[\texttt{k}] \leq \texttt{a}[\texttt{k}']$$

We'll rename this invariant `partitioned`(a, i), and add the predicate to our code.

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

This is pertinent to our main invariant, because bubble sort works by maintaining a partition on `a`: the first half is unsorted, whereas the second half is sorted, and per our new invariant, contains values that are at least as large as those in the unsorted portion. We add this invariant to both loops, as is a property of the outer loop that is preserved by the inner loop.

```
method BubbleSort(a: array<int>)
  modifies a
  requires a != null
  ensures sorted(a, 0, a.Length-1)
{
  var i := a.Length - 1;
  while(i > 0)
    invariant -1 <= i < a.Length
    invariant sorted(a, i, a.Length-1)
    invariant partitioned(a, i)
  {
    var j := 0;
    while(j < i)
      invariant 0 < i < a.Length && 0 <= j <= i
      invariant sorted(a, i, a.Length-1)
      invariant partitioned(a, i)
    {
      if(a[j] > a[j+1])
      {
        a[j], a[j+1] := a[j+1], a[j];
      }
      j := j + 1;
    }
    i := i - 1;
  }
}
```

After attempting to verify the above code, it appears that our invariants are still not strong enough. The verifier's error message tells us that the two non-basic outer invariants might not be preserved by the loop. Let's examine the relevant basic path, and its corresponding verification condition to gain further clarity.

```
invariant 0 < i < a.Length && 0 <= j <= i &&
          sorted(a, i, a.Length-1) &&
          partitioned(a, i)
assume i <= j;
i := i - 1;
invariant -1 <= i < a.Length &&
          sorted(a, i, a.Length-1) &&
          partitioned(a, i)
```

The corresponding VC, after a bit of simplification, is:

$$0 < \texttt{i} < \texttt{a.Length} \wedge \texttt{i} = \texttt{j} \wedge \texttt{sorted}(\texttt{a},\ \texttt{i},\ \texttt{a.Length} - 1) \wedge \texttt{partitioned}(\texttt{a},\ \texttt{i}) \\ \rightarrow \texttt{sorted}(\texttt{a},\ \texttt{i} - 1,\ \texttt{a.Length} - 1) \wedge \texttt{partitioned}(\texttt{a},\ \texttt{i} - 1)$$

This invariant is not valid, and the problem is similar to what we encountered earlier when we tried to generalize by accepting an invariant that widened the sorted region. Consider the following counterexample, which satisfies the antecedent but not the consequent.

$$\begin{array}{ccccccc} 2 & & 1 & & 4 & & 3 & & 5 \\ & & & & & & \textit{(j)} & & \textit{(i)} \end{array}$$

However, a bit of thought leads us to conclude that bubble sort would never produce this counterexample state. We know this because bubble sort always propagates the *largest* value from the unsorted region to the bottom of the sorted region. Indeed, throughout the inner loop $\texttt{a}[\texttt{j}]$ contains the largest value processed so far. Formalized, the corresponding invariant is relevant to the inner loop:

$$\forall \texttt{k}.0 \leq \texttt{k} \leq j \rightarrow \texttt{a}[\texttt{k}] \leq \texttt{a}[\texttt{j}]$$

After adding this, we have an implementation and proof that Dafny "almost" accepts.

```
method BubbleSort(a: array<int>)
  modifies a
  requires a != null
  ensures sorted(a, 0, a.Length-1)
{
  var i := a.Length - 1;
  while(i > 0)
    invariant -1 <= i < a.Length
    invariant sorted(a, i, a.Length-1)
    invariant partitioned(a, i)
  {
    var j := 0;
    while(j < i)
      invariant 0 < i < a.Length && 0 <= j <= i
      invariant sorted(a, i, a.Length-1)
      invariant partitioned(a, i)
      invariant forall k :: 0 <= k <= j ==> a[k] <= a[j]
    {
      if(a[j] > a[j+1])
      {
        a[j], a[j+1] := a[j+1], a[j];
      }
      j := j + 1;
    }
    i := i - 1;
  }
}
```

At this point, the verifier no longer complains about the loop invariants being preserved. Rather, it notifies us that the postcondition might not hold on a path through our method, and points to the head of the loop as the possible location at which our code may be incorrect (or our proof incomplete). What is going on here? The loop exit condition conjoined with our key loop invariant appears sufficient to establish the postcondition:

$$\texttt{sorted(a, i, a.Length} - 1) \wedge \texttt{i} \leq 0 \rightarrow \texttt{sorted(a, 0, a.Length} - 1)$$

Informally, we know that when the loop exits, `i` is either 0 or -1. When $\texttt{i} = 0$, the relevant VC is certain to hold:

$$\texttt{sorted(a, i, a.Length} - 1) \wedge \texttt{i} = 0 \rightarrow \texttt{sorted(a, 0, a.Length} - 1)$$

When $\texttt{i} = -1$, then we know that $\texttt{a.Length} = 0$, and our `sorted` predicate becomes:

$$\begin{array}{ll} & \forall i, j.0 \leq \texttt{l} \leq \texttt{i} \leq \texttt{j} \leq \texttt{u} < -1 \rightarrow \texttt{a[i]} \leq \texttt{a[j]} \\ \Leftrightarrow & \mathit{false} \rightarrow \texttt{a[i]} \leq \texttt{a[j]} \end{array}$$

So it would seem that the postcondition holds in either case. It seems that the verifier is not picking up on the relationship we need,

$$\texttt{i} < 0 \rightarrow \texttt{a.Length} = 0$$

This is clearly true before the outer loop executes (i.e., immediately after `i`) is assigned, and cursory inspection of the loop body suggests that it is maintained until the loop terminates. We can add it as an invariant to our outer loop, and the verifier accepts our proof.

```
method BubbleSort(a: array<int>)
  modifies a
  requires a != null
  ensures sorted(a, 0, a.Length-1)
{
  var i := a.Length - 1;
  while(i > 0)
    invariant i < 0 ==> a.Length == 0
    invariant -1 <= i < a.Length
    invariant sorted(a, i, a.Length-1)
    invariant partitioned(a, i)
  {
    var j := 0;
    while(j < i)
      invariant 0 < i < a.Length && 0 <= j <= i
      invariant sorted(a, i, a.Length-1)
      invariant partitioned(a, i)
      invariant forall k :: 0 <= k <= j ==> a[k] <= a[j]
    {
      if(a[j] > a[j+1])
      {
        a[j], a[j+1] := a[j+1], a[j];
      }
      j := j + 1;
    }
    i := i - 1;
  }
}
```

Strengthening the Specification. We have managed to prove that when `BubbleSort` terminates, its parameter contains a sorted list whenever it begins non-null. However, our specification is not as strong as we may like it to be for some purposes. For example, consider the following program, which satisfies the specifications we've given to `BubbleSort`.

```
method BubbleSort(a: array<int>)
  modifies a
  requires a != null
  ensures sorted(a, 0, a.Length-1)
{
  var i := a.Length - 1;
  while(i >= 0)
    invariant -1 <= i < a.Length
    invariant i >= 0 ==> forall j :: i < j < a.Length ==> a[j] == a[a.Length-1]
    invariant sorted(a, i+1, a.Length-1)
  {
    a[i] := a[a.Length-1];
    i := i - 1;
  }

}
```

While a well-intentioned programmer is unlikely to deviate in spirit so far from the contract, the point remains that our specification does not require that the contents of `a` differ in the post-state only in their order from the pre-state; this is crucial to our expectations of what a sorting procedure does.

We can strengthen our specification by adding to the postcondition that the final contents of *a* be a *permutation* of their initial contents. Dafny has a syntactic facility for referring to the initial contents of a reference type in specifications: the `old(a)` notation.

```
method BubbleSort(a: array<int>)
  modifies a
  requires a != null
  ensures sorted(a, 0, a.Length-1)
  ensures permutation(a[..], old(a[..]))
```

This specification says that when `BubbleSort` terminates, the new contents of `a` must be a permutation of their initial contents. Notice that when we refer to `a` in the arguments of `permutation`, we suffix with [..]. This is Dafny's notation for computing an *array slice*, which returns the contents of an array between specified indices as a `seq`. Because we do not provide any indices, this returns the entire contents of `a` as a `seq<int>`, and we do this so that the `permutation` predicate we write can take `seq<int>` arguments rather than `array<int>`; sequences are nicer to work with in specifications than arrays. We could have computed a sequence corresponding to any range of `a` by writing `a[low..high]` had we needed to refer to specific portions of `a`.

Now we must write a predicate for `permutation`. Intuitively, we know that an array `a` is a permutation of `b` if and only if for each possible value `v`, the number of occurrences of `v` in `a` is the same as the number of occurrences in `b`. So, to encode the right predicate we will need a function that counts the number of occurrences of a particular value in a sequence.

```
function count(a: seq<int>, v: int): nat
{
  if(|a| > 0) then
    if(a[0] == v) then 1 + count(a[1..], v)
    else count(a[1..], v)
  else 0
}
```

With this function in hand, we can write our predicate for `permutation` in a straightforward manner.

```
predicate permutation(a: seq<int>, b: seq<int>)
{
  forall v :: count(a, v) == count(b, v)
}
```

When we attempt to verify our method against the new specification, Dafny tells us that the post-condition might not hold. Of course, it is referring to the postcondition mentioning `permutation`, which means that our current annotations are not sufficient to establish the newly-strengthened specification. We must obtain new annotations sufficient to prove this property.

Unlike in the case of `sorted`, our implementation does not gradually establish that the current `a` is a permutation of `old(a)`. So, adding an invariant that incrementally asserts that a larger portion of the array is a permutation of the old may not be necessary, and in fact may be more difficult to prove than the straightforward choice of:

```
invariant perm(old(a[..]), a[..])
```

Intuitively, the only way in which our code modifies the contents of `a` is by swapping adjacent pairs of elements. If we place this invariant, which is clearly strong enough to prove the postcondition in question, on both loops, then we can reasonably expect to prove it by reasoning about localized portions of `a` near the point of our modifications inside the inner loop. However, Dafny does not accept our annotations yet.

Although it may seem obvious to us that no elements are added or removed from `a` in the line:

```
a[j], a[j+1] := a[j+1], a[j];
```

Dafny does not know how to connect these manipulations of the array to the properties relevant for `permutation`. We need to demonstrate that the contents of `a` immediately after the update are a permutation of the contents before it. To do this, it will be helpful to have some way of referring to both versions of `a` simultaneously. Dafny supports *ghost variables* for exactly this purpose. Ghost variables are declared with the modifier `ghost`, and are used only at verification time; they are not compiled into variables that are allocated and updated when the program is run, so we do not need to worry about the impact on performance that manipulating these variables would otherwise have.

We declare and initialize a new ghost variable to track the contents of `a` before the update:

```
ghost var a' := a[..];
a[j], a[j+1] := a[j+1], a[j];
```

We now need to demonstrate that `permutation`$(\texttt{a}[..], \texttt{a}')$ holds after the update:

```
ghost var a' := a[..];
a[j], a[j+1] := a[j+1], a[j];
assert permutation(a[..], a');
```

The verifier still is not convinced that we've proven the claim, although we can check to make sure that this assertion is sufficient to establish our loop invariants by temporarily switching the `assert` to an `assume`:

```
ghost var a' := a[..];
a[j], a[j+1] := a[j+1], a[j];
assume permutation(a[..], a');
```

When we do this, the loop invariants and postconditions successfully verify, so we know that we're on the right track. This is an effective proof debugging technique that should be used frequently to ensure that we aren't wasting time proving facts that are irrelevant to our specifications.

Perhaps the verifier is having difficulty reasoning monolithically about the entirety of $\texttt{a}$ and $\texttt{a}'$. To help it along, let's break both values down into simpler parts that highlight their differences. We know that the only positions in which the two differ are $\texttt{j}, \texttt{j}+1$. So, it should be the case that

$$\texttt{a}[..\texttt{j}] = \texttt{a}'[..\texttt{j}] \wedge \texttt{a}[\texttt{j}+2..] = \texttt{a}'[\texttt{j}+2..]$$

A quick check with an `assert` verifies that this is the case, and is recognized by Dafny. However, we don't need this assertion by itself in our proof, so we remove it before moving on.

Decomposing $\texttt{a}$ and $\texttt{a}'$ in this manner makes it obvious that the two are permutations of eachother: $\texttt{a}[..\texttt{j}]$ and $\texttt{a}'[..\texttt{j}]$ are permutations of eachother, $\texttt{a}[\texttt{j}+2..]$ and $\texttt{a}'[\texttt{j}+2..]$ are as well, and finally $\texttt{a}[\texttt{j}..\texttt{j}+2]$ and $\texttt{a}'[\texttt{j}..\texttt{j}+2]$ are as well. However, Dafny does not accept this final fact, so we make it very clear by being explicit about the contents of $\texttt{a}[\texttt{j}..\texttt{j}+2], \texttt{a}'[\texttt{j}..\texttt{j}+2]$:

```
ghost var a' := a[..];
a[j], a[j+1] := a[j+1], a[j];
ghost var v1, v2 := a[j], a[j+1];
assert a[..] == a[..j] + [v1, v2] + a[j+2..];
assert a' == a[..j] + [v2, v1] + a[j+2..];
assert permutation([v1, v2], [v2, v1]);
assume permutation(a[..], a');
```

The only line that the verifier will not accept is the final. It seems that we have done enough to prove the desired property, but let's take a closer look to understand where Dafny might be getting tripped up. `permutation` is a predicate over `count`, and we expect that $\texttt{a}$ is a permutation of $\texttt{a}'$ because for every value, `count` is the same in $\texttt{a}[..\texttt{j}]$ and $\texttt{a}'[..\texttt{j}]$, $\texttt{a}[\texttt{j}..\texttt{j}+2]$ and $\texttt{a}'[\texttt{j}..\texttt{j}+2]$, and $[\texttt{v1}, \texttt{v2}], [\texttt{v2}, \texttt{v1}]$. Because $\texttt{a}$ and $\texttt{a}'$ are composed of only these parts, we expect that `count` on the full contents will match up as well. We can check this intuition with an assertion:

```
assert forall v :: count(a[..], v) == count(a[..j], v) + count([v1, v2], v) +
  count(a[j+2..], v);
```

The verifier complains, so presumably this is our culprit. Dafny does not realize that `count` is distributive across sequence concatenation, i.e., that:

$$\forall \mathtt{v}.\mathtt{count}(\mathtt{a}+\mathtt{b}+\mathtt{c},\mathtt{v}) = \mathtt{count}(\mathtt{a},\mathtt{v}) + \mathtt{count}(\mathtt{b},\mathtt{v}) + \mathtt{count}(\mathtt{c},\mathtt{v})$$

Now that we know this is the issue, it is not particularly surprising: `count` is non-trivial function over sequences, and we require quantified reasoning over its behavior. We will need to prove it, and because it is a general property that does not depend on the specifics of `BubbleSort`, we will do so by creating a lemma that can potentially be reused later.

```
lemma count_distrib3(a: seq<int>, b: seq<int>, c: seq<int>)
  ensures forall v :: count(a + b + c, v) ==
                        count(a, v) + count(b, v) + count(c, v)
```

Before we begin proving the main argument that `count` is distributive across concatenation, we observe that the signature of this lemma may be more complicated than we'd like. The context in which we'd like to use the lemma needs to relate this property to the concatenation of three sequences, but intuition should tell us that we can achieve this by sequencing assertions of distributivity with respect to pairs of sequences. Formally, this corresponds to the claim that:

$$\begin{aligned}\forall \mathtt{a},\mathtt{b},\mathtt{c},\mathtt{v}.\quad & (\mathtt{count}(\mathtt{a}+\mathtt{b},\mathtt{v}) = \mathtt{count}(\mathtt{a},\mathtt{v}) + \mathtt{count}(\mathtt{b},\mathtt{v}) \wedge \\ & \mathtt{count}((\mathtt{a}+\mathtt{b})+\mathtt{c},\mathtt{v}) = \mathtt{count}(\mathtt{a}+\mathtt{b},\mathtt{v}) + \mathtt{count}(\mathtt{c},\mathtt{v})) \\ & \rightarrow \mathtt{count}(\mathtt{a}+\mathtt{b}+\mathtt{c},\mathtt{v}) = \mathtt{count}(\mathtt{a},\mathtt{v}) + \mathtt{count}(\mathtt{b},\mathtt{v}) + \mathtt{count}(\mathtt{c},\mathtt{v})\end{aligned}$$

To reason in this way, we need a lemma `count_distrib2` with the signature:

```
lemma count_distrib2(a: seq<int>, b: seq<int>)
  ensures forall v :: count(a + b, v) == count(a, v) + count(b, v)
```

This allows us to complete `count_distrib3` by invoking `count_distrib2` twice:

```
lemma count_distrib3(a: seq<int>, b: seq<int>, c: seq<int>)
  ensures forall v :: count(a + b + c, v) ==
                        count(a, v) + count(b, v) + count(c, v)
{
  count_distrib2(a, b);
  count_distrib2(a + b, c);
}
```

Note that we could have used `count_distrib2` directly in the body of `BubbleSort` to prove the claim we're after, but it would have cluttered the proof with more calls to the lemma.

Now we need to prove the postcondition for `count_distrib2`. This is a universally-quantified claim over all sequence element values `v`, so we'll start off with the `forall` statement in the body, which allows us to write a postcondition that will hold for every value in the iteration range of the statement:

```
lemma count_distrib2(a: seq<int>, b: seq<int>)
  ensures forall v :: count(a + b, v) == count(a, v) + count(b, v)
{
  forall(v: int)
    ensures count(a + b, v) == count(a, v) + count(b, v)
  {

  }
}
```

Copying the formula within the scope of our postcondition's quantifier is appropriate here, and sufficient to prove the claim. Although it's fairly straightforward to see in this example, we can test it by inserting an `assume false` statement in the body of the loop.

We must now decide how to proceed with the main proof of the distributive property. Because the objects that we refer to in the lemma postcondition are sequences, it is worth taking a moment to consider the set of possible values that these objects can take. Sequences can be defined inductively, with the empty sequence [ ] serving as the base case. The definition is as follows:

$$\texttt{seq<T>} \quad ::= \quad [\,] \mid [\texttt{T}] + \texttt{seq<T>}$$

This suggests that we can use structural induction to prove our claim. We will induce on the structure of `a`, starting with the base case of `a` = [ ]:

```
lemma count_distrib2(a: seq<int>, b: seq<int>)
  ensures forall v :: count(a + b, v) == count(a, v) + count(b, v)
{
  forall(v: int)
    ensures count(a + b, v) == count(a, v) + count(b, v)
  {
    if(a == []) {
      calc == {
        count(a + b, v);
          { assert a + b == b; }
        count(b, v);
        0 + count(b, v);
          { assert count(a, v) == 0; }
        count(a, v) + count(b, v);
      }
    }
  }
}
```

We use an equality calculation to demonstrate the claim, and proceed by using the fact that `a`'s emptiness makes concatenation with `b` the identity. The last few steps of this calculation are not necessary for Dafny to accept the claim in this case, but are provided here to make the proof explicit.

Continuing on with the inductive case, we need to separate `a` into its constituent parts according. Having done so, we can recursively invoke the lemma on the smaller constituents to prove the claim.

```
lemma count_distrib2(a: seq<int>, b: seq<int>)
  ensures forall v :: count(a + b, v) == count(a, v) + count(b, v)
{
  forall(v: int)
    ensures count(a + b, v) == count(a, v) + count(b, v)
  {
    if(a == []) {
      calc == {
        count(a + b, v);
          { assert a + b == b; }
        count(b, v);
        0 + count(b, v);
          { assert count(a, v) == 0; }
        count(a, v) + count(b, v);
      }
    } else {
      calc == {
        count(a + b, v);
          { assert a + b == [a[0]] + (a[1..] + b); }
        count([a[0]] + (a[1..] + b), v);
        (if a[0] == v then 1 + count(a[1..] + b, v) else count(a[1..] + b, v));
          { count_dist(a[1..], b); }
        (if a[0] == v then 1 + count(a[1..], v) + count(b, v)
         else count(a[1..], v) + count(b, v));
        count(a, v) + count(b, v);
      }
    }
  }
}
```

Notice that the third line comes directly from the definition of `count`. As before, several of the final steps are not necessary for Dafny to accept our proof, but are provided here for documentation.

This completes the proof that `count` distributes across concatenation, and thus the proof of `BubbleSort`. The complete listing of `BubbleSort`'s code is given in the next figure.

```
method BubbleSort(a: array<int>)
  modifies a
  requires a != null
  ensures sorted(a, 0, a.Length-1) && perm(old(a[..]), a[..])
{
  var i := a.Length - 1;
  while(i > 0)
    invariant i < 0 ==> a.Length == 0
    invariant sorted(a, i, a.Length-1)
    invariant part(a, i)
    invariant perm(old(a[..]), a[..])
  {
    var j := 0;
    while(j < i)
      invariant 0 <= j <= i
      invariant sorted(a, i, a.Length-1)
      invariant part(a, i)
      invariant forall k :: 0 <= k <= j ==> a[k] <= a[j]
      invariant perm(old(a[..]), a[..])
    {
      if(a[j] > a[j+1]) {
        ghost var a' := a[..];
        a[j], a[j+1] := a[j+1], a[j];
        ghost var v1, v2 := a[j], a[j+1];
        assert a[..] == a[..j] + [v1, v2] + a[j+2..];
        assert a' == a[..j] + [v2, v1] + a[j+2..];
        count_distrib3(a[..j], [v1, v2], a[j+2..]);
        count_distrib(3a[..j], [v2, v1], a[j+2..]);
        assert forall v :: count(a[..], v) ==
              count(a[..j], v) + count([v1, v2], v) + count(a[j+2..], v);
        assert perm([v1, v2], [v2, v1]);
        assert perm(a[..], a');
      }
      j := j + 1;
    }
    i := i - 1;
  }
}
```

Appendix: Using Z3 to check verification conditions

When reasoning about the validity of verification conditions on basic paths, it can be tedious to look for counterexamples or attempt validity proofs by hand. Here, we'll see how to use Z3, the decision procedure utilized by Dafny, directly to assist our reasoning.

Running Z3 You will find the binaries for Z3 in the Dafny distribution you downloaded earlier (assuming that you did not opt for the Visual Studio extension). You can also use Z3 via the web interface at `http://rise4fun.com/z3`.

To run Z3 from the command line, invoke its binary (`z3`) with a single parameter corresponding to a file containing the SMT problem that you would like to solve. Z3 accepts several input formats, and the examples that we'll see in this section are in the SMT2 format that you can read more about at `http://smtlib.cs.uiowa.edu/`.

Encoding VCs To demonstrate the use of Z3 to encode verification conditions, we'll look at all of the partial correctness conditions in our completed `BubbleSort` method. However, to make the examples relatively simpler and easier to follow, we'll look at the version of `BubbleSort` that doesn't prove `permutation(old(a[..]), a[..])`.

The first basic path proceeds from the method entry point to the beginning of the outer loop.

```
1 requires a != null
2 i := a.Length - 1
3 invariant -1 <= i < a.Length
4 invariant sorted(a, i, a.Length-1)
5 invariant partitioned(a, i)
```

To encode this in Z3, we'll first declare symbols corresponding to our syntactic entities. Starting with the variables `a`, `i`, we have the following.

```
1 (declare-const a (Array Int Int))
2 (declare-const i Int)
```

We declare variables using `declare-const`, which accepts a type argument. `a` is defined to be an element from the domain of Z3's theory of arrays taking integer indices to integer-valued elemented, and `i` and integer. The `Array` types used by Z3 have the same axioms as the theory of arrays that we studied earlier in the semester.

```
1 (declare-const a (Array Int Int))
2 (declare-const i Int)
```

Now we need to define the predicates `sorted` and `partitioned`. Recall that predicates are functions ranging over Booleans, so we use `declare-fun` to declare them, and then assert their semantics as axioms (i.e., universally-quantified logical sentences). We also need some way of referring to the length of our arrays, as the theory of arrays does not account for it, but it is mentioned in our formulas. To do so, we'll introduce an uninterpreted function `length` ranging over arrays, and assert an axiom that requires the length of any array to be at least zero.

```
(declare-const a (Array Int Int))
(declare-const i Int)

(declare-fun sorted ((Array Int Int) Int Int) Bool)
(declare-fun partitioned ((Array Int Int) (Int)) Bool)
(declare-fun length ((Array Int Int)) Int)

(assert (forall ((a (Array Int Int)))
  (<= 0 (length a))))

(assert (forall ((a (Array Int Int)) (l Int) (u Int))
  (iff
    (sorted a l u)
    (forall ((i Int) (j Int))
      (=>
        (and (<= 0 l) (<= l i) (<= i j) (<= j u) (< u (length a)))
        (<= (select a i) (select a j)))))))

(assert (forall ((a (Array Int Int)) (i Int))
  (iff
    (partitioned a i)
    (forall ((k1 Int) (k2 Int))
      (=>
        (and (<= 0 k1) (<= k1 i) (< i k2) (< k2 (length a)))
        (<= (select a k1) (select a k2)))))))
```

Before moving on, notice a few things about this input format. Z3 is an SMT solver that is fundamentally based on the "lazy" propositional encoding algorithm that we studied earlier in the semester. Having algorithmic roots in DPLL, you should expect that it accepts CNF formulas over some first-order background theory as input, and returns `sat`, `unsat`, or `unknown` as its result (`unknown` is necessary as the background theory may be undecidable, or the user may want to place a hard limit on the amount of time the solver is given). The lines beginning with `assert` correspond to the CNF clauses that we're giving as our input, and as such, they take a Boolean-valued term (i.e., formula) as their only argument. When we ask Z3 to solve our instance, it will attempt to decide whether each asserted formula can be satisfied conjunctively. However, it is worth pointing out that Z3 is somewhat flexible with respect to the contents of each clause. It does not strictly require the input to be a CNF, so the clauses don't necessarily need to be disjunctive; they can contain arbitrary logical connectives and theory symbols, as long as the final result is a Boolean type.

The second thing to notice is the fact that formulas and terms are given in **prefix notation**. That is, rather than writing:

$$\texttt{F}_1 \texttt{ and } \texttt{F}_2 \texttt{ and } \cdots \texttt{ and } \texttt{F}_n$$

Z3 expects its input to be given as:

$$(\texttt{and } \texttt{F}_1 \texttt{ F}_2 \cdots \texttt{ F}_n)$$

If you haven't seen this notation before, it may take a litle getting used to. However, the basic idea is straightforward: rather than placing the operator between its arguments, it is placed at the beginning of a parenthetical list, and the arguments follow in-order. In general, when a binary operator is associative, Z3 will allow you to list an arbitrary number of arguments rather than nesting successive instances of the operator within the arguments. This is why we were able to use

one instance of `and` in the example above, rather than:

$$(\texttt{and}\ \texttt{F}_1\ (\texttt{and}\ \texttt{F}_2\ (\texttt{and}\ \cdots\ \texttt{F}_n)))$$

Getting back to our example, recall that we'd like to check the verification condition for the basic path:

```
requires a != null
i := a.Length - 1
invariant -1 <= i < a.Length
invariant sorted(a, i, a.Length-1)
invariant partitioned(a, i)
```

We've encoded our variables and all of the necessary predicates by making universally-quantified assertions that reflect their semantics. For example, recalling the predicate that we wrote in Dafny for `sorted`:

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

We encoded this in Z3 using the following assertion:

```
(assert (forall ((a (Array Int Int)) (l Int) (u Int))
  (iff
    (sorted a l u)
    (forall ((i Int) (j Int))
      (=>
        (and (<= 0 l) (<= l i) (<= i j) (<= j u) (< u (length a)))
        (<= (select a i) (select a j)))))))
```

This translation is fairly straightforward, with the only noteworthy artifact being the fact that we universally-quantified the arguments to the predicate when encoding it in Z3. This reflects the fact that we want this definition to hold for any arguments that we might apply the predicate to. We did not account for the precondition in the Dafny predicate, as we have not encoded the possibility that the array reference might be null into our SMT satisfiability problem. When Dafny produces verification conditions, it certainly does account for this, but because we only reason about sortedness in situations where the array reference is non-null (recall the precondition on `BubbleSort`), we don't need to address this additional complexity at the moment.

Inspecting the above basic path, we'll replace the precondition $\texttt{a} \neq \texttt{null}$ with *true*. This gives us the verification condition:

$$\begin{aligned} & \mathit{true} \rightarrow \mathsf{wlp}(\texttt{i} := |\texttt{a}| - 1, -1 \leq \texttt{i} < |\texttt{a}|) \wedge \texttt{sorted}(\texttt{a}, \texttt{i}, |\texttt{a}| - 1) \wedge \texttt{partitioned}(\texttt{a}, \texttt{i})) \\ \Leftrightarrow\ & -1 \leq |\texttt{a}| - 1 < |\texttt{a}| \wedge \texttt{sorted}(\texttt{a}, |\texttt{a}| - 1, |\texttt{a}| - 1) \wedge \texttt{partitioned}(\texttt{a}, |\texttt{a}| - 1) \end{aligned}$$

We'd like to use Z3, a satisfiability solver, to check the validity of this formula. Recall that we can exploit the connection between satisfiability and validity to accomplish this. Namely, we will check

to determine whether the negation of our formula is satisfiable:

$$\neg(-1 \leq |\mathtt{a}| - 1 < |\mathtt{a}| \wedge \mathtt{sorted}(\mathtt{a}, |\mathtt{a}| - 1, |\mathtt{a}| - 1) \wedge \mathtt{partitioned}(\mathtt{a}, |\mathtt{a}| - 1))$$

Continuing from the Z3 instance we left off with before, we simply add another assertion corresponding to this condition:

```
(declare-const a (Array Int Int))
(declare-const i Int)

(declare-fun sorted ((Array Int Int) Int Int) Bool)
(declare-fun partitioned ((Array Int Int) (Int)) Bool)
(declare-fun length ((Array Int Int)) Int)

(assert (forall ((a (Array Int Int)))
  (<= 0 (length a))))

(assert (forall ((a (Array Int Int)) (l Int) (u Int))
  (iff
    (sorted a l u)
    (forall ((i Int) (j Int))
      (=>
        (and (<= 0 l) (<= l i) (<= i j) (<= j u) (< u (length a)))
        (<= (select a i) (select a j)))))))

(assert (forall ((a (Array Int Int)) (i Int))
  (iff
    (partitioned a i)
    (forall ((k1 Int) (k2 Int))
      (=>
        (and (<= 0 k1) (<= k1 i) (< i k2) (< k2 (length a)))
        (<= (select a k1) (select a k2)))))))

(assert (not (and
               (<= -1 (- (length a) 1))
               (< (- (length a) 1) (length a))
               (sorted a (- (length a) 1) (- (length a) 1))
               (partitioned a (- (length a) 1)))))

(check-sat)
```

The last line instructs Z3 to decide the satisfiability of our assertions. If we save our instance to `vc.smt2`, then we invoke Z3 as follows:

```
$ z3 vc.smt2
```

It should return promptly with:

```
$ z3 vc.smt2
unsat
```

Which tells us that our verification condition is valid: its negation is unsatisfiable.

Now let's take a look at an invalid VC. Recall from earlier that the basic path:

```
invariant 0 < i < a.Length && 0 <= j <= i &&
          sorted(a, i, a.Length-1) &&
          partitioned(a, i)
assume i <= j;
i := i - 1;
invariant -1 <= i < a.Length &&
          sorted(a, i, a.Length-1) &&
          partitioned(a, i)
```

With corresponding VC:

$$0 < \texttt{i} < \texttt{a.Length} \wedge \texttt{i} = \texttt{j} \wedge \texttt{sorted}(\texttt{a},\ \texttt{i},\ \texttt{a.Length} - 1) \wedge \texttt{partitioned}(\texttt{a},\ \texttt{i}) \\ \rightarrow \texttt{sorted}(\texttt{a},\ \texttt{i} - 1,\ \texttt{a.Length} - 1) \wedge \texttt{partitioned}(\texttt{a},\ \texttt{i} - 1)$$

is invalid. We can use Z3 to check this by deciding the satisfiability of:

$$0 < \texttt{i} < \texttt{a.Length} \wedge \texttt{i} = \texttt{j} \wedge \texttt{sorted}(\texttt{a},\ \texttt{i},\ \texttt{a.Length} - 1) \wedge \texttt{partitioned}(\texttt{a},\ \texttt{i}) \\ \wedge \neg(\texttt{sorted}(\texttt{a},\ \texttt{i} - 1,\ \texttt{a.Length} - 1) \wedge \texttt{partitioned}(\texttt{a},\ \texttt{i} - 1))$$

Translating the problem into Z3's input format, assuming that we've already defined the relevant variables and axioms:

```
(assert (< 0 i))
(assert (< i (length a)))
(assert (= i j))
(assert (sorted a i (- (length a) 1)))
(assert (partitioned a i))
(assert (not (and (sorted a (- i 1) (- (length a) 1)) (partitined a (- i 1)))))

(check-sat)
```

It appears that Z3 hangs on this instance. Adding a 10-second timeout to the solver's configuration at the top of the input file:

```
(set-option :timeout 10000)
```

We see that the solver returns `unknown`. Because program verification is one of the primary applications that Z3 was developed to enable, it has been engineered to identify `unsat` instances very quickly, as these correspond to valid verification conditions. However, for invalid conditions, which correspond to `sat` instances, it is oftentimes not as effective. When Dafny's verifier sees an `unknown` answer from Z3, it assumes that the VC is invalid and returns an error message. The program locations that it identifies in its error message are generated by obtaining a "reason" for the unknown answer, which Z3 is usually able to return in the form of a subset of assertions on which it had difficulty reasoning.

In our case, the difficulty is due to the quantifiers that we used to define `sorted` and `partitioned`; as we've discussed previously, quantified formulas are especially difficult for automated reasoning techniques to handle. Nonetheless, Dafny and Z3 do remarkably well handling such instances in a large number of cases. A significant part of the challenge in developing a verifier lies in finding a good SMT encoding for the generated verification conditions. The exact encoding that Dafny

uses differs from the one we used here in numerous ways, which is why the verifier did not respond with a timeout error message when we used the wrong loop invariant corresponding to this basic path. You can examine the Z3 verification conditions generated by Dafny by invoking it with the `proverLog` argument:

```
$ mono Dafny.exe prog.dfy /proverLog:@PROC@-vc.smt2
```

This will cause the compiler to generate a file for each verification condition, where `@PROC@` expands to the procedure that the condition corresponds to. As an exercise, examine the verification conditions generated for `BubbleSort`, and try to make the Z3 verification conditions we wrote above correspond more closely to those generated by Dafny. Understanding, at least in some level of detail, how the verifier works behind the scenes will further remove the mystery from some of the error messages that you encounter when proving your code.