Software Testing, Quality Assurance and Maintenance Winter 2026

Lecture 21 — March 23, 2026

Patrick Lam *version 1*

We are still pretty confident in our bubble sort implementation. We did, however, use multisets. This is not a correctness issue, but it's good to know how to implement things ourselves.

Using sequences rather than multisets

We are instead going to use sequences, which we briefly mentioned in Lecture 16. As we (briefly) saw back there, Dafny can seamlessly convert arrays into sequences; we'll use sequences in our lemmas. A Dafny seq⟨T⟩ is an immutable sequence of elements of type T; for example, seq⟨int⟩ is a sequence of ints. Unlike arrays and lists, but more like a Java String, they are immutable. Recall that if 'a' is an array then 'a[..]' is a sequence with the contents of the array.

Let's start with some helper functions. Note that since a seq is a functional datatype, we can't modify it, but we can easily take a sub-sequence with a[1..].

```
/**
  Returns number of occurrences of a given value 'val' inside a given sequence
 */
function count(a:seq<int>, val:int): nat
{
  if |a| = 0 then 0
  else
    if a[0] = val then 1 + count(a[1..], val)
    else count(a[1..], val)
}
```

Dafny knows that this function terminates—the length of count is decreasing in recursive calls.

We use this function to define a predicate which says whether a is a permutation of b—recall that this was the main problem in our incomplete specification of bubble sort.

```
/**
  A sequence 'b' is a permutation of sequence 'a' if every integer 'v' has
  exactly the same number of occurrences in 'a' and in 'b'
 */
ghost predicate perm(a: seq<int> , b:seq<int>)
{
  ∀ v • count(a, v) = count(b, v)
}
```

Dafny complains that perm is not compilable unless we label it a ghost predicate, which means that we can use it in specifications but not in code. That's all we need anyway.

We know some things about permutation. Dafny can figure them out, but we help it by expressing them.

```
/** Permutation is transitive.
    Dafny can prove that automatically.
 */
lemma trans_perm(a:seq<int>, b:seq<int>, c:seq<int>)
  requires perm(a, b) ∧ perm(b, c)
  ensures perm(a, c)
{}

/** Permutation is commutative.
    Dafny can prove that automatically.
  */
lemma comm_perm(a:seq<int> , b:seq<int> )
  requires perm(a, b)
  ensures perm(b, a)
{}
```

Here's a lemma about swapping two positions in a sequence. Dafny cannot prove it automatically. I'll leave this as an exercise.

```
/**
   If sequences `a` and `b` differ only in two positions, and the values in
   these to positions are swapped, then sequences are permutations of one
   another.

   This is assumed without a proof. Completing the proof is left as an exercise.
 */
lemma swap_lemma(a: seq<int>, b:seq<int>, pos1:nat, pos2:nat)
  requires |a| = |b|
  requires 0 ≤ pos1 < pos2 < |a|
  requires ∀ k • 0 ≤ k < |a| ∧ k ≠ pos1 ∧ k ≠ pos2 ⟹ a[k] = b[k]
  requires a[pos1] = b[pos2]
  requires a[pos2] = b[pos1]
  ensures perm(a, b)
  { assume(false); }
```

We use the following method in our bubble sort. Here we specify it in isolation:

```
/** A helper method to swap two positions in an array */
method swap(a: array<int>, pos1:nat, pos2:nat)
  modifies a
  requires 0 ≤ pos1 < pos2 < a.Length
  ensures ∀ k • 0 ≤ k < a.Length ∧ k ≠ pos1 ∧ k ≠ pos2 ⟹ a[k] = old(a[k])
  ensures a[pos1] = old(a[pos2])
  ensures a[pos2] = old(a[pos1])
{
  var temp := a[pos2];
  a[pos2] := a[pos1];
  a[pos1] := temp;
}
```

Now here is an implementation of bubble sort using the helper method and the lemmas. We do a lot of the work for Dafny here—Dafny doesn't know these properties of permutations by default.

```
/**
   Verified version of bubble sort, in which the prove that the output is
   permutation of the input is done explicitly without using any special help
   from Dafny.
 */
method bubbleSort3 (a: array<int>)
  requires a.Length > 0
  ensures ∀ u, v • 0 ≤ u < v < a.Length ⟹ a[u] ≤ a[v]
  ensures perm(a[..], old(a[..]))
  modifies a
{
  var i:nat := 1;

  while (i < a.Length)
    invariant i ≤ a.Length
    invariant ∀ u, v • 0 ≤ u < v < i ⟹ a[u] ≤ a[v]
    invariant perm(a[..], old(a[..]))
  {
    var j:nat := i;
    while (j > 0)
      invariant 0 ≤ j ≤ i
      invariant ∀ u, v • 0 ≤ u < v < j ⟹ a[u] ≤ a[v]
      invariant ∀ u, v • 0 ≤ u < j < v ≤ i ⟹ a[u] ≤ a[v]
      invariant ∀ u, v • j ≤ u < v ≤ i ⟹ a[u] ≤ a[v]
      invariant perm(a[..], old(a[..]))
    {
      ghost var old_a := a[..];
      if (a[j-1] > a[j]) {
        // Swap a[j-1] and a[j]
        swap(a, j-1, j);
        swap_lemma(old_a, a[..], j-1, j);
        comm_perm(old_a, a[..]);
      }
      j := j - 1;

      trans_perm(a[..], old_a, old(a[..]));
      assert(perm(a[..], old(a[..])));
    }
    i := i+1;
  }
}
```

Here we add the postcondition we were missing before in our naïve initial try. The invariants are also similar to the previous non-refactored example, except that we use perm instead of the multisets that we had invoked before. The swap operation itself, though, also uses the swap_lemma and the commutativity property comm_perm. Finally, we also point out to Dafny that perm is transitive at the end of the outer loop.

It turns out, though, that Dafny actually only really needs the swap_lemma.

Bounding runtimes

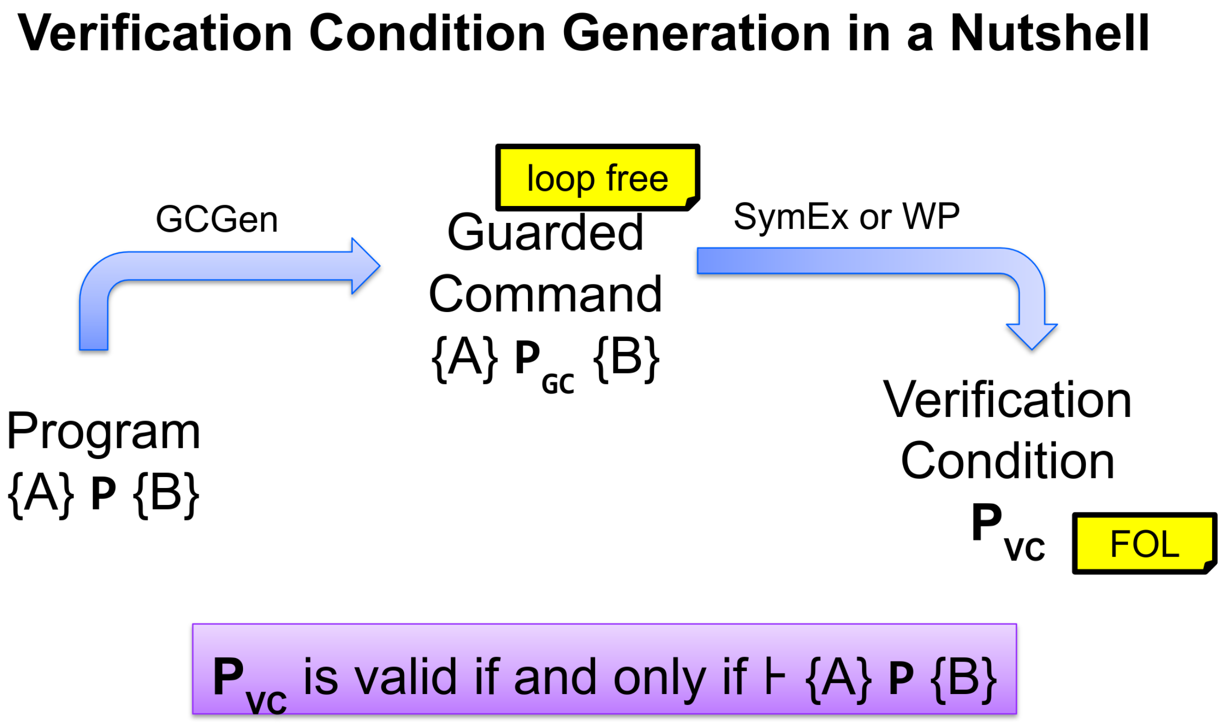
Dafny can do that too: it can prove that this bubble sort implementation is $O(|a|^2)$. We add some instrumentation to our earlier implementation and assert on the instrumentation's values.

```
method bubbleSort4 (a: array<int>)
  requires a.Length > 0
  ensures ∀ u, v • 0 ≤ u < v < a.Length ⟹ a[u] ≤ a[v]
  modifies a
{
  // counter that counts number of executions of the main loop
  var cnt := 0;
  var i:nat := 1;

  while (i < a.Length)
    invariant i ≤ a.Length
    invariant ∀ u, v • 0 ≤ u < v < i ⟹ a[u] ≤ a[v]
    invariant cnt ≤ a.Length * i
  {
    var j:nat := i;
    while (j > 0)
      invariant 0 ≤ j ≤ i
      invariant ∀ u, v • 0 ≤ u < v < j ⟹ a[u] ≤ a[v]
      invariant ∀ u, v • 0 ≤ u < j < v ≤ i ⟹ a[u] ≤ a[v]
      invariant ∀ u, v • j ≤ u < v ≤ i ⟹ a[u] ≤ a[v]
      invariant cnt ≤ a.Length * i + (i-j)
    {
      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;
      // increment count each iteration of the loop
      cnt := cnt + 1;
    }
    i := i+1;
  }
  // claim that cnt is bounded above by |a|^2
  assert(cnt ≤ a.Length * a.Length);
}
```

Verification Condition Generation

Moving on, and continuing on the implementing-things-ourselves theme, we are going to talk more about how Dafny works under the hood—earlier we had talked about loops, now we'll talk about normal statements. This is called *verification condition generation*. Here's a picture, which should not be a surprise.



Step 1: Loop-Free Guarded Commands

For verification purposes, we first compile the program into loop-free guarded commands. Here's the grammar for these commands.

```
c ::= assume b
  |   assert b
  |   havoc x
  |   c1; c2
  |   c1 || c2
```

The guards are the asserts and assumes. You can't execute a program once it's been compiled to guarded commands, but proving correctness of the guarded commands is equivalent to proving correctness of the original.

Let's look at the rules to translate WHILE into guarded commands. This is similar to Hoare Logic.

- GC(skip): assume true
- GC(x := e): assume tmp = x; havoc x; assume (x = e[tmp/x]), where tmp is fresh.
- GC($c_1; c_2$): GC(c_1); GC(c_2)
- GC(if b then c_1 else c_2): (assume b; GC(c_1) || assume $\neg b$; GC(c_2)
- GC(while b inv I do c): ...

The rule for assignment is like the one for Hoare Logic. We zero out x and then assume that it is e, taking care to not use the x that we're defining if it should occur in e, but rather old x.

For if, it looks like we have parallel composition, but in any actual execution, either b is concretely true or concretely false. So one of the branches has what amounts to assume false and verification would stop on that branch in that condition. The execution continues on the branch that does not have an assume false.

We've also seen the translation for loops:

- GC(while b inv I do c):

```
        assert I;
        havoc $x_1$; ...; havoc $x_n$;        // (variables modified in c)
        assume I;
        ((assume $b$; GC(c); assert I; assume false) ||
           assume $\neg b$)
```

Example. Let's look at how this works on our multiplication example again.

```
{n >= 0}
p := 0;
x := 0;
while x < n inv p = x * m && x <= n do
  x := x + 1;
  p := p + m
{p = n * m}
```

The conversion is mechanical and we get this:

```
{ n >= 0 }
assume p0 = p; havoc p; assume p = 0;
assume x0 = x; havoc x; assume x = 0;
assert p = x * m && x <= n;
havoc x; havoc p; assume p = x * m && x <= n;
  ((assume x < n;
    assume x1 = x; havoc x; assume x = x1 + 1;
    assume p1 = p; havoc p; assume p = p1 + m;
    assert p = x * m && x <= n; assume false)
   || assume x >= n)
{ p = n * m }
```

For the assignment statements, we can see both the cases where there is no capture and where an assignment refers to the old value of, for instance, `x`—it introduces temporary variable `x1`. For the while loop, we assert the invariant before the loop, then clear everything but the invariant, assume it, and show it at the end of the loop body. When exiting the loop, we assume the negation of the loop body.

Step 2: Verification Condition Generation

Here are two ways we could proceed.

Idea 1: Exhaustive symbolic execution of GC program. Symbolic execution sometimes appears in this course, but I've removed it this year. Basically, instead of tracking concrete values of program variables, you track abstract (symbolic) values. It is, in principle, possible to use symbolic execution to verify the guarded commands: start at the beginning, keep on tracking the symbolic state, and see if we can prove the postcondition at the end. The program is correct if no assertion can fail.

The symbolic execution is implicitly constructing verification conditions and bringing them through to the end.

Because of the way this works, we start with the pre-condition, and use the program structure, but here we are bringing *everything* through to the end, even if that is much more than what is needed to prove the postcondition. The symbolic state, for instance, can get exponentially huge, and we'd rather not have that. Anyway, we won't talk about this more.

Idea 2: Propagate the post-condition backwards through the program. Here, the idea is to use the post-condition and proceed backwards, checking that the required pre-condition, plus the program behaviour, is strong enough to ensure the post-condition.

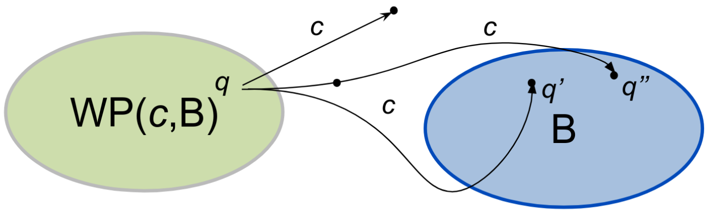
That is, for a Hoare triple $\{A\}P\{B\}$, we generate FOL formula $A \Rightarrow F(P, B)$, where backwards propagation $F(P, B)$ is formalized in terms of *weakest preconditions*.

This approach has always seemed less intuitive to me than the symbolic execution approach, but it does work better.

The weakest precondition $\mathrm{WP}(c, B)$ holds for any state q whose c-successor states all satisfy B; that is,

$$q \models \mathrm{WP}(c, B) \quad \text{iff} \quad \forall q' \in Q.\ q \xrightarrow{c} q' \Rightarrow q' \models B$$

Or, as a picture:



It really is backwards, but effective.

WP Examples. Let's again work through some examples. This is similar to the Hoare Logic examples we saw earlier. It's a mechanical process to start with, but then also we can simplify the formulas.

$$\begin{array}{lll} \{ \qquad\qquad \} & x := z + w & \{x \geq y\} \\ \{ \qquad\qquad \} & \textsf{if } y > 0 \textsf{ then } x := x + 1 \textsf{ else } x := y + 4 & \{x \geq 42\} \\ \{ \qquad\qquad \} & \textsf{if } y > 0 \textsf{ then } x := z \textsf{ else } x := y & \{x \geq y + z\} \end{array}$$

For the first one, we push through the RHS to obtain weakest precondition

$$z + w \geq y$$

If you think about it for a bit—or, better yet, work through some examples—you can see that anything weaker is not going to give the needed postcondition, but that what we have here does.

We can mechanically push backwards $x \geq 42$ in the second case, getting a disjunction:

$$(y > 0 \wedge x + 1 \geq 42) \vee (y \leq 0 \wedge y + 4 \geq 42)$$

which simplifies to

$$(y > 0 \wedge x \geq 41) \vee (y \leq 0 \wedge y \geq 38)$$

and we can drop the second disjunct, which can never be true, to get weakest precondition:

$$y > 0 \land x \geq 41$$

Similarly, in the third case, the mechanical process yields:

$$(y > 0 \land z \geq y + z) \lor (y \leq 0 \land y \geq y + z)$$

which simplifies to

$$(y > 0 \land 0 \geq y) \lor (y \leq 0 \land 0 \geq z)$$

also known as

$$(y = 0) \lor (y \leq 0 \land z \leq 0)$$

This is all mechanical and Dafny does it for you.

So, now we can compute weakest pre-conditions. How does that help for program verification? We have a method, along with a post-condition, and compute the weakest pre-condition. If the method pre-condition is stronger than the weakest pre-condition, then any valid way that you call the method is going to ensure the stated post-condition. This verifies the method.

(Another way of saying this is that, if you put the pre-condition as an assume at the beginning of the method, then you have to compute the weakest pre-condition as true; otherwise, anything that is "left over" is a way to break the function contract.)

Computing Weakest Pre-conditions. Here are rules for mechanically computing the weakest preconditions.

- WP(assume b, B): $b \Rightarrow B$
- WP(assert b, B): $b \land B$
- WP(havoc x, B): $B[a/x]$ (a fresh in B; "replace x by a")
- WP(c_1 ; c_2, B): WP(c_1, WP(c_2, B))
- WP(c_1 || c_2, B): WP(c_1, B) $\land$ WP(c_2, B)

Let's start with c_1 || c_2. In that case, the weakest precondition has to be the conjunction of those for c_1 and c_2, because either of those (or both, in some sense) could execute, so we'd better account for them.

Conversely, for sequential composition, we calculate the WP for c_2 first, and then take its output, go backwards, and compute the weakest precondition for c_1. So you can get to the start of c_1 by going through c_2 and then c_1.

For a havoc statement, we wipe out everything we knew about x and put in some fresh a to replace it. B has to be true no matter what x is afterwards. The new a can pick up more constraints as the computation continues backwards through the method.

The weakest precondition for an assert is that the expression b being asserted must hold as part of the weakest precondition. Otherwise, the execution aborts. Note that this doesn't change the state.

Finally, for an assume statement, we use the property of implication where we make no guarantees about what happens if the antecedent is false. So, if the antecedent is true (the assumption holds), then we say the incoming thing B; otherwise, we basically quietly abort the execution and pretend nothing happened. (There is a subtle point about partial correctness here; the requirement is just that there *exists* an execution that has the desired effect, not that all executions do, so assume can drop executions and it's fine.)

Putting everything together. This is what Dafny does:

Given a Hoare triple $H \equiv \{A\}P\{B\}$,

- compute $c_H :=$ assume A; $\mathrm{GC}(P)$; assert B;
- compute $\mathrm{VC}_H := \mathrm{WP}(c_H, \mathsf{true})$; and,
- prove $\vdash \mathrm{VC}_H$ using a theorem prover.

One last example. But it's a bit of a doozy. Let's compute the weakest precondition of the multiplication program, which we'd already converted to the guarded command language.

```
WP( assume n ≥ 0;
    assume p0 = p; havoc p; assume p = 0;
    assume x0 = x; havoc x; assume x = 0;
    assert p = x * m ∧ x  ≤  n;
    havoc x; havoc p; assume p = x * m ∧ x  ≤  n;
        ((assume x < n;
          assume x1 = x; havoc x; assume x = x1 + 1;
          assume p1 = p; havoc p; assume p = p1 + m;
          assert p = x * m ∧ x  ≤  n; assume false)
        || assume x ≥ n) ;
    assert p = n * m, true)
```

We have incorporated the precondition and postcondition already, as an assume/assert pair, so we aim to finish with weakest precondition true.

We start at the end with the assert statement, which got tacked on to the end using sequential composition. WP for the assert is `p = n * m` and applying the rule for `;` gives us new B being the same expression:

```
WP( assume n ≥ 0;
    assume p0 = p; havoc p; assume p = 0;
    assume x0 = x; havoc x; assume x = 0;
    assert p = x * m ∧ x ≤ n;
    havoc x; havoc p; assume p = x * m ∧ x ≤ n;
        ((assume x < n;
          assume x1 = x; havoc x; assume x = x1 + 1;
          assume p1 = p; havoc p; assume p = p1 + m;
          assert p = x * m ∧ x ≤ n; assume false)
        || assume x ≥ n), p = n * m)
```

Continuing backwards, the next sequential composition is between

```
  assume p = x * m ∧ x ≤ n
```

and what we got converting the while body, so we split this and compute the weakest preconditon of the loop body separately; it'll be the input to computing WP on everything before that.

```
  WP( assume n ≥ 0;
      assume p0 = p; havoc p; assume p = 0;
      assume x0 = x; havoc x; assume x = 0;
      assert p = x * m ∧ x ≤ n;
      havoc x; havoc p; assume p = x * m ∧ x ≤ n,
        WP((assume x < n;
            assume x1 = x; havoc x; assume x = x1 + 1;
            assume p1 = p; havoc p; assume p = p1 + m;
            assert p = x * m ∧ x ≤ n; assume false)
          || assume x ≥ n, p = n * m))
```

We have a ||, so we change that into a conjunction.

```
  WP( assume n ≥ 0;
      assume p0 = p; havoc p; assume p = 0;
      assume x0 = x; havoc x; assume x = 0;
      assert p = x * m ∧ x ≤ n;
      havoc x; havoc p; assume p = x * m ∧ x ≤ n,
        WP(assume x < n;
            assume x1 = x; havoc x; assume x = x1 + 1;
            assume p1 = p; havoc p; assume p = p1 + m;
            assert p = x * m ∧ x ≤ n; assume false, p = n * m)
      ∧ WP(assume x ≥ n, p = n * m))
```

Let's convert the last assume into an implication.

```
  WP( assume n ≥ 0;
      assume p0 = p; havoc p; assume p = 0;
      assume x0 = x; havoc x; assume x = 0;
      assert p = x * m ∧ x ≤ n;
      havoc x; havoc p; assume p = x * m ∧ x ≤ n,
        WP(assume x < n;
            assume x1 = x; havoc x; assume x = x1 + 1;
            assume p1 = p; havoc p; assume p = p1 + m;
            assert p = x * m ∧ x ≤ n; assume false, p = n * m)
      ∧ x ≥ n ⇒ p = n * m)
```

Now for the loop body, where we have another sequential composition with its last assume false:

```
WP( assume n ≥ 0;
    assume p0 = p; havoc p; assume p = 0;
    assume x0 = x; havoc x; assume x = 0;
    assert p = x * m ∧ x ≤ n;
    havoc x; havoc p; assume p = x * m ∧ x ≤ n,
      WP(assume x < n;
          assume x1 = x; havoc x; assume x = x1 + 1;
          assume p1 = p; havoc p; assume p = p1 + m;
          assert p = x * m ∧ x ≤ n; WP(assume false, p = n * m))
    ∧ x ≥ n ⇒ p = n * m)
```

Let's process the assume false and understand how that stops considering what happens going off the end of the loop body.

```
WP( assume n ≥ 0;
    assume p0 = p; havoc p; assume p = 0;
    assume x0 = x; havoc x; assume x = 0;
    assert p = x * m ∧ x ≤ n;
    havoc x; havoc p; assume p = x * m ∧ x ≤ n,
      WP(assume x < n;
          assume x1 = x; havoc x; assume x = x1 + 1;
          assume p1 = p; havoc p; assume p = p1 + m;
          assert p = x * m ∧ x ≤ n, false ⇒ p = n * m)
    ∧ x ≥ n ⇒ p = n * m)
```

So now B for the WP for the loop body is false ⇒ . . .: we don't care what happens after the end of the loop body, we just need to show all the assertions before the end of the loop. That is,

```
WP( assume n ≥ 0;
    assume p0 = p; havoc p; assume p = 0;
    assume x0 = x; havoc x; assume x = 0;
    assert p = x * m ∧ x ≤ n;
    havoc x; havoc p; assume p = x * m ∧ x ≤ n,
      WP(assume x < n;
          assume x1 = x; havoc x; assume x = x1 + 1;
          assume p1 = p; havoc p; assume p = p1 + m;
          assert p = x * m ∧ x ≤ n, true)
    ∧ x ≥ n ⇒ p = n * m)
```

Continuing inside the loop body and applying sequential composition and assert,

```
WP( assume n ≥ 0;
    assume p0 = p; havoc p; assume p = 0;
    assume x0 = x; havoc x; assume x = 0;
```

```
      assert p = x * m ∧ x ≤ n;
      havoc x; havoc p; assume p = x * m ∧ x ≤ n,
        WP(assume x < n;
            assume x1 = x; havoc x; assume x = x1 + 1;
            assume p1 = p; havoc p; assume p = p1 + m,
            p = x * m ∧ x ≤ n)
      ∧ x ≥ n ⇒ p = n * m)
```

We apply another assume:

```
  WP( assume n ≥ 0;
      assume p0 = p; havoc p; assume p = 0;
      assume x0 = x; havoc x; assume x = 0;
      assert p = x * m ∧ x ≤ n;
      havoc x; havoc p; assume p = x * m ∧ x ≤ n,
        WP(assume x < n;
            assume x1 = x; havoc x; assume x = x1 + 1;
            assume p1 = p; havoc p,
            p = p1 + m ⇒ p = x * m ∧ x ≤ n)
      ∧ x ≥ n ⇒ p = n * m)
```

Now for the havoc and assume, we introduce fresh variable pa1 and substitute it for p:

```
  WP( assume n ≥ 0;
      assume p0 = p; havoc p; assume p = 0;
      assume x0 = x; havoc x; assume x = 0;
      assert p = x * m ∧ x ≤ n;
      havoc x; havoc p; assume p = x * m ∧ x ≤ n,
        WP(assume x < n;
            assume x1 = x; havoc x; assume x = x1 + 1,
            p1 = p ∧ pa1 = p1 + m ⇒ pa1 = x * m ∧ x ≤ n)
      ∧ x ≥ n ⇒ p = n * m)
```

Continuing with x = x1 + 1:

```
  WP( assume n ≥ 0;
      assume p0 = p; havoc p; assume p = 0;
      assume x0 = x; havoc x; assume x = 0;
      assert p = x * m ∧ x ≤ n;
      havoc x; havoc p; assume p = x * m ∧ x ≤ n,
        WP(assume x < n;
            assume x1 = x; havoc x,
            x = x1 + 1 ∧ p1 = p ∧ pa1 = p1 + m ⇒ pa1 = x * m ∧ x ≤ n)
      ∧ x ≥ n ⇒ p = n * m)
```

and another havoc, introducing fresh `xa1`:

```
WP( assume n ≥ 0;
    assume p0 = p; havoc p; assume p = 0;
    assume x0 = x; havoc x; assume x = 0;
    assert p = x * m ∧ x ≤ n;
    havoc x; havoc p; assume p = x * m ∧ x ≤ n,
      WP(assume x < n; assume x1 = x,
          xa1 = x1 + 1 ∧ p1 = p ∧ pa1 = p1 + m ⇒ pa1 = xa1 * m ∧ xa1 ≤ n)
    ∧ x ≥ n ⇒ p = n * m)
```

which leaves us with another `assume x1 = x` to handle; we use ∧ because we are tacking it onto the antecedent of the implication here (essentially, we are simultaneously assuming multiple things).

```
WP( assume n ≥ 0;
    assume p0 = p; havoc p; assume p = 0;
    assume x0 = x; havoc x; assume x = 0;
    assert p = x * m ∧ x ≤ n;
    havoc x; havoc p; assume p = x * m ∧ x ≤ n,
      WP(assume x < n,
          x1 = x ∧ xa1 = x1 + 1 ∧ p1 = p ∧ pa1 = p1 + m ⇒ pa1 = xa1 * m ∧ xa1 ≤ n)
    ∧ x ≥ n ⇒ p = n * m)
```

and finally completing the loop body:

```
WP( assume n ≥ 0;
    assume p0 = p; havoc p; assume p = 0;
    assume x0 = x; havoc x; assume x = 0;
    assert p = x * m ∧ x ≤ n;
    havoc x; havoc p; assume p = x * m ∧ x ≤ n,
      x < n ∧ x1 = x ∧ xa1 = x1 + 1 ∧ p1 = p ∧ pa1 = p1 + m
           ⇒ pa1 = xa1 * m ∧ xa1 ≤ n
    ∧ x ≥ n ⇒ p = n * m)
```

If we continue through the beginning of the method, we get this formula:

```
n ≥ 0 ∧ p0 = p ∧ pa3 = 0 ∧ x0 = x ∧ xa3 = 0 ⇒
  pa3 = xa3 * m ∧ xa3 ≤ n ∧
  (pa2 = xa2 * m ∧ xa2 ≤ n ⇒
    ((xa2 < n ∧ x1 = xa2 ∧ xa1 = x1 + 1 ∧ p1 = pa2 ∧ pa1 = p1 + m) ⇒
        pa1 = xa1 * m ∧ xa1 ≤ n
    ) ∧
    (xa2 ≥ n ⇒ pa2 = n * m)
  )
```

which is equivalent to the conjunction of the three formulas, representing the three things to prove for a loop:

$$n \geq 0 \wedge p0 = p \wedge pa3 = 0 \wedge x0 = x \wedge xa3 = 0 \Rightarrow pa3 = xa3 * m \wedge xa3 \leq n$$

$$n \geq 0 \wedge p0 = p \wedge pa3 = 0 \wedge x0 = x \wedge xa3 = 0 \wedge pa2 = xa2 * m \wedge xa2 \leq n \Rightarrow (xa2 \geq n \Rightarrow pa2 = n * m)$$

$$n \geq 0 \wedge p0 = p \wedge pa3 = 0 \wedge x0 = x \wedge xa3 = 0 \wedge pa2 = xa2 * m \wedge xa2 \leq n \wedge x1 = xa2 \wedge xa1 = x1 + 1 \wedge p1 = pa2 \wedge pa1 = p1 + m \Rightarrow pa1 = xa1 * m \wedge xa1 \leq n$$

with the first condition being that the loop invariant holds upon entry; the second that it implies the postcondition; and the third that it is preserved by the loop body.

It is possible to simplify these constraints to (or just feed them as-is to a solver):

$$n \geq 0 \Rightarrow 0 = 0 * m \wedge 0 \; leq \; n$$

$$xa2 \leq n \wedge xa2 \geq n \Rightarrow xa2 * m = n * m$$

$$xa2 < n \Rightarrow xa2 * m + m = (xa2 + 1) * m \wedge xa2 + 1 \leq n$$

which are all valid, proving that the original Hoare triple was also valid.