Proofs of Correctness: Loop Invariants

COMP 221 — Suhas Arehalli

Showing that our algorithm is correct for a particular case is straightforward — we figure out what the correct answer should be for that particular input and make sure our algorithm provides that answer. This is what we might consider a *unit test* — just like the kind you saw in prior courses — but for pseudocode. However, there are typically an unbounded number of possible inputs for an algorithm, and thus no way to write a test for every case one-by-one!

Our goal in this unit is to get comfortable with a couple of techniques for proving correctness for non-trivial programs. The first, *loop invariants*, help us handle iteration (i.e., for/while loops).

1 Preliminaries

Before we get started proving correctness, we need to figure out what correctness means, formally. This takes the form of a **problem statement** or **definition** — a formal description of a problem's inputs (including any **pre-conditions** about them that we can assume) and a formal description of what the output must look like. For example, we can think about the Array Search problem:

Problem Statement (Array Search)**.**
Input: An Array $A[1 \dots n]$ and an element e
Output: i such that $A[i] = e$ or $NULL$ if $\forall 1 \leq i \leq n$, $A[i] \neq e$.

And consider a Linear Search algorithm like the following:

```
function LinearSearch(Array A[1...n], Element e)
    for i ← 1 to n do
        if A[i] = e then
            return i
        end if
    end for
    return NULL
end function
```

Here we can use the problem statement's formality to prove that we will always return the correct answer!

Statement 1.
LinearSearch solves the Array Search Problem (i.e., LinearSearch is correct).

Proof. Break out into cases by the return value of LinearSearch.

Case 1: LinearSearch returns some i. The only such return statement is within the if-statement that checks whether $A[i] = e$, and thus if we return some i, that i satisfies $A[i] = e$, as desired.

Case 2: LinearSearch returns $NULL$. This only occurs after the while loop, and if we've reached this line, this means that we never entered the if-statement. This implies that for each $1 \leq i \leq n$, $A[i] \neq e$, as desired.[1] □

However, one we get to trickier problems and algorithms, we'll need to build stronger tools to reason about correctness. Consider the Sorting Problem

Problem Statement (Sorting)**.**
Input: An Array $A[1 \dots n]$
Output: An Array $B[1 \dots n]$ with the same elements as A such that $B[1] \leq B[2] \leq \cdots \leq B[n]$.

This is one of the most important problems — practically and historically — in the algorithms curriculum!

For convenience, we're typically not going to sweat the *with the same elements* portion of the description, as the algorithms that we consider are only going to modify our array by swapping the positions of elements, which naturally will preserve the elements sorted in the array. We'll instead focus on making sure the second condition is met — that our algorithms end with a *sorted* array.

2 Loop Invariants

An iterative algorithm is one that relies on loops constructs (for and while loops, for instance). Oftentimes, each loop iteration makes a small amount of progress toward our goal. For instance, the sorting algorithms we're going to look at will typically make our working array closer and closer to sorted with each iteration. Loop invariants are simply a way to mathematically formalize the idea of making that incremental progress. In brief, our scheme goes as follows:

1. We state our **loop invariant** — a mathematical claim that states that after n iterations, some specific amount of progress (dependent on n!) toward our goal is made.
2. We prove that the claim is true when $n = 0$ — that is, before any iterations are made, our statement holds true.
3. We prove that each iteration maintains the truth of our loop invariant: As n increases (i.e., we go through more iterations), our invariant should tell us more progress must be made. We need to show that the new iteration makes that progress!
4. We prove that once the loop terminates, whatever number of iterations we went through is enough to prove the algorithm's correctness.

These are sometimes referred to as the **loop invariant**, the **initialization step**, the **maintenance step**, and the *termination step*. However, this should remind you a bit of the structure of *induction*: We build a predicate $P(n)$, prove $P(0)$ — the base case — and then that $P(k)$ (or

[1] If we want to be *really pedantic*, we can build this out with a loop invariant using the technique we'll see below: After the iteration where $i = k$, $\forall 1 \leq j \leq k$, $A[j] \neq e$. However, this is a simple enough case that we won't need to pull out these powerful techniques.

$P(0), \ldots, P(k)$ if it's strong induction) implies $P(k+1)$ — the inductive step. All that's missing is the termination step — showing that the statement we get when we halt the induction/iteration is enough to prove the algorithm correct!

But before we get to the more complex cases, let's survey some simple examples.

3 Example: Sums

Consider the following summation algorithm:

```
function SUM(Array A[1...n])
    k ← 0
    for i ← 1 to n do
        k ← k + A[i]
    end for
    return k
end function
```

Which we posit is a solution to the summation problem

Problem Statement (Summation)**.**
Input: Array $A[1 \ldots n]$
Output: $\sum_{i=1}^{n} A[i]$

Here, we can observe that our progress is measured by the state of the variable k, which, every iteration, will increment by another element from A. We can formalize this via a loop invariant like

Statement 2.
After the nth iteration of our loop, $k = \sum_{i=1}^{n} A[i]$

Proof. Proceed by induction.

Base Case: Before the first iteration ($n = 0$), we have that $k = 0 = \sum_{i=1}^{0} A[i]$ (we sum 0 elements).

Inductive Step: Assume as our inductive hypothesis that after the nth iteration, $k = \sum_{i=1}^{n} A[i]$. On the $n+1$st iteration, we have that $k \leftarrow k + A[n+1]$, and thus at the end of the iteration, $k = \sum_{i=1}^{n} + A[n+1] = \sum_{i=1}^{n+1} A[i]$ as desired. □

And from this, we need only move from the loop invariant being true to show our summation algorithm is correct:

Corr. 1 (SUM is correct)**.**
SUM with input $A[1 \ldots n]$ will return $\sum_{i=1}^{n} A[i]$

Proof. As the for loop runs for n iteration, our loop invariant tells us that after our for-loop, $k = \sum_{i=1}^{n} A[i]$, which we return. □

Take a moment to study the structure and format of this proof, as it intentionally does not use any techniques other than loop invariants and induction. Our next example is more sophisticated, and will require some comfort with both the idea of a loop invariant, analyzing code, and the other proof techniques at our disposal!

4 Example: Insertion Sort

Consider the sorting algorithm below, called *insertion sort*:

```
function InsertionSort(Array A[1...n])
    for i ← 2 to n do
        j ← i
        while j > 1 and A[j] < A[j − 1] do
            swap(A[j], A[j − 1])
            j ← j − 1
        end while
    end for
    return A
end function
```

For convenience, let's re-write this into two function so we can focus on one loop at a time:

```
function Insert(Array A[1...m])
    j ← m
    while j > 1 and A[j] < A[j − 1] do
        swap(A[j], A[j − 1])
        j ← j − 1
    end while
end function
function InsertionSort(Array A[1...n])
    for i ← 2 to n do
        A[1...i] ←Insert(A[1...i])
    end for
    return A
end function
```

Take a moment and get familiar with how this algorithm works. Just run through the code!

What you should note is that Insert does what the name implies: Takes an element, $A[m]$, and inserts it to the sorted position into an already sorted subarray $A[1 \dots m-1]$. We can formalize this into a lemma and prove it via a loop invariant!

Lemma 1 (Insert)**.**
Given $A[1 \dots m]$ such that $A[1 \dots m-1]$ is sorted, INSERT($A[1 \dots m]$) is sorted.

Before we get to the proof, let's establish the loop invariant:

Statement 3.
After the ith iteration of the while loop in INSERT (ending with $j = m - i$), $A[m-i \dots m]$ and $A[1 \dots m-i-1]$, and $f \leq e, \forall e \in A[m-i+1 \dots m], f \in A[1 \dots m-i-1]$.

Verify that this is true on a handful of examples! Our intuition is that after this iteration, we haven't messed with $A[1 \dots m-i-1]$ so it remains sorted and we've swapped our unsorted element — $A[m-i]$ — over in a way that makes sure all the (already sorted) elements moved to it's right are greater than it. The last bit just reiterates that the elements to the right of $A[m-i]$ are greater than those the the left. Let's now prove the lemma formally!

Proof. We'll prove this by first showing the loop invariant **??** is true via induction.

Base Case: Before the first iteration ($i = 0$), we must show that $A[m-i \dots m] = A[m \dots m]$ is sorted (trivial, as it only contains one element!), that $A[1 \dots m-i-1] = A[1 \dots m-1]$ is sorted (as given), and that $f \in A[1 \dots m]$ is less that all 0 elements in $A[m+1 \dots m]$ (trivial!).

Inductive Step: Suppose after iteration $i = k$, our loop invariant holds, and thus $A[m-k \dots m]$ and $A[1 \dots m-k-1]$ are both sorted and $A[m-k-1] \leq A[m-k+1]$

After one more iteration ($i = k+1$), if we have not already terminated, we know that (since $j = m - i$), $A[m-k] < A[m-k-1]$ and we swap them. Denote the post swap array as A'. After the swap, we have that $A'[m-k-1] < A'[m-k]$. Now, since we began the iteration with $A[m-k-1] \leq A[m-k+1]$, post-swap we have that $A'[m-k] \leq A'[m-k+1]$, and since we began with $A[m-k \dots m]$ sorted, we have (post-swap), that

$$A'[m-k-1] \leq A'[m-k+1] \leq \dots \leq A[m]$$

Ignore the first inequality and chain it with the prior inequalities we established to get

$$A'[m-k-1] \leq A[m-k] \leq A[m-k+1] \leq \dots \leq A[m]$$

Which tells us that $A'[m-k-1 \dots m] = A[m-(k+1) \dots m]$ is sorted. Then observe that we started with $A[1 \dots m-k-1]$ sorted, and thus

$$\begin{aligned} A[1] \leq \dots A[m-k-2] &\leq A[m-k-1] \\ A'[1] \leq \dots A'[m-k-2] &\leq A'[m-k] \\ A'[1] \leq \dots A'[m-(k+1)-1] &\leq A'[m-(k+1)+1] \end{aligned}$$

Which is the same as saying $A'[1 \dots m-(k+1)-1]$ is sorted, and that $A[m-(k+1)-1] \leq A[m-(k+1)+1]$.

Taken together, we've shown that $A[1 \dots m-(k+1)-1]$ is sorted, $A[m-(k+1) \dots m]$ is sorted, and that $A[m-(k+1)-1] \leq A[m-(k+1)+1]$. Note that that with the knowledge that

Before:	a_1	...	a_{m-i-1}	a_{m-i}	a_{m-i+1}	...	a_m
After:	a_1	...	a_{m-i}	a_{m-i--1}	a_{m-i+1}	...	a_m
	1	...	$m-i-1$	$m-i$	$m-i+1$	...	m

Figure 1: A visualization of the ith iteration of the INSERT algorithm. Blue elements are those that began sorted, with the deeper blues indicating larger values. The Orange element is the one being inserted into the sorted subarray. We swap when we realize that $a_{m-i-1} > a_{m-i}$. Translate all of the fancy array notation to this diagram to confirm that (1) the IH is true in the "Before" row and that the loop invariant for the ith iteration is satisfied in the "After" row.

the first two subarrays are sorted, the last inequality tells us the largest element of $A[1 \dots m-(k+1)-1]$ is less than or equal to the smallest element of $A[m-(k+1)+1 \dots m]$, and thus $f \leq e$ for $e \in A[1 \dots m-(k+1)-1], f \in A[m-(k+1)+1 \dots m]$ — our loop invariant for our current iteration, $i = k+1$.

Thus, by induction, our loop invariant holds.

We then note that our loop terminates under two conditions:

Case 1: If $j = 1$, we have completed $i = m-1$ iterations. Thus, applying the loop invariant, we have that $A[m-(m-1) \dots m] = A[1 \dots m] = A$ is sorted, as desired.[2]

Case 2: If $A[j] > A[j-1]$ for some value of j, note that we have undergone $i = m-j$ iterations, and thus we know via our loop invariant that $A[1 \dots m-(m-j)-1] = A[1 \dots j-1]$ is sorted and $A[m-(m-j) \dots m] = A[j \dots m]$ is sorted.[3] Combine these three inequalities to find that

$$A[1] \leq \cdots \leq A[j-1] \leq A[j] \leq \cdots \leq A[m]$$

Or, equivalently, that A is sorted. □

Note that this proof is (1) long and a bit tedious and (2) captures the key mechanics of the insert operation. Take a moment to glance at Fig. **??** if you'd like a visualization of what our inductive step of loop invariant statement is doing — it's a lot of indexing and inequalities, but you should confirm that all of the statements are, in fact, natural properties of the array as INSERT iterates through the array.

But, before it gets too tedious, let's finish up a proof of the full insertion sort's correctness using this lemma

Statement 4 (Insertion sort is correct)**.**
On any array input A, INSERTIONSORT returns a sorted array.

Via the loop invariant:

Statement 5.
After the kth iteration (when $i = k+1$), $A[1 \dots k+1]$ is sorted

Proof. Proceed by induction

Base Case: After 0 iterations, $A[1 \dots 1]$ contains a single element and is thus trivially sorted.

Inductive Step: Assume as an inductive hypothesis that after k iterations, our invariant holds and $A[1 \dots k+1]$ is sorted. Then, on the next, $k+1$st, iteration, this $A[1 \dots k+2]$ is passed to INSERT. Since $A[1 \dots k+1]$ is sorted via our inductive hypothesis, by Lemma **??**, we have that $A[1 \dots k+2]$ is sorted by the end of the iteration as desired. □

It's worth noting that the lemma is the vast majority of the work here — once we establish that INSERT works, the fact that Insertion Sort makes progress is clear!