Proofs of Correctness

Suhas Arehalli

COMP221 - Spring 2024

1 Why Proofs of Correctness

- So far, we've been pretty informal about our algorithms being right
  - Sometimes we do this through testing! Other times we trust our intuition.
  - We spent a lot of time talking about how to formally show how *fast* our algorithms are, so it's probably important to show that our algorithms are *correct* as well.
  - This will become more important as our algorithms get more and more complex. Our intuitions may fail us!
- To prove algorithms are correct, we can often rely on some tools you picked up in Discrete Math. The most important such tool is *mathematical induction*.

2 Loop Invariants and Induction

- Consider Insertion Sort (Alg. 1):

Algorithm 1 Insertion sort, but more condensed than before

```
function InsertionSort(Array A)
    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
end function
```

- Remind yourself of how insertion sort works: at the beginning of the kth iteration of the outer loop, $A[1...k-1]$ is sorted, and $A[k]$ may be out of place. After the kth iteration, we've *inserted* the value at $A[k]$ (through the while loop) into it's correct position, and so $A[1...k]$ is sorted.

- The insertion of insertion sort happens in the inner while loop — we swap until what began at $A[K]$ is at $A[j]$, which should be it's correct position.

- Let's also talk about *problem specification*: What is it that a sorting algorithm must do?
  - In general, a sorting algorithm takes as input an array A and returns an array A' that is *sorted and* contains the same elements as A.
  - The version of insertion sort I've given you here is specific kind of sort called an *in-place* sort where in practice, we return nothing and simply modify A such that at the end of the algorithm, A is sorted. For simplicity, let's call A at the start of our in-place sort B and A at the end of our in-place sort B'. We need to prove that B' is sorted ($B[1] \leq B[2] \leq \cdots \leq B[N]$) and that B and B' contain the same elements (Formally, that $e \in B'$ if and only if $e \in B$)
  - Given that insertion sort can only swap the positions of elements in the array it manipulates, it's not possible for B and B' to contain different elements! This is both simpler and less interesting to show than the fact that insertion sort properly orders the elements in A, so we'll just leave the proof as a sketch:
    - Show that swapping 2 elements does not introduce new elements or remove elements (If $e \in A$ before a swap, $e \in A$ after. If $e \notin A$ before a swap, $e \notin A$ after).
    - Then, if we're being really rigorous, prove by induction that no number of swaps will ever violate this property.
  - This leaves us to show that B' (i.e., A at the end of insertion sort) is *sorted*.

- Now here is our general strategy for proving iterative algorithms correct:
  1. Find a **Loop Invariant**: A statement that we will prove is true after every iteration of a loop in our algorithm.
     (a) Often this will be in terms of the index of the loop, as we'll see for insertion sort.
  2. Then we'll prove the loop invariant to be true by induction.
     (a) Our base case will typically be showing that our invariant is true *before* the first iteration of the loop. People sometimes refer to this as as the *initialization* step, I'll try to use the base case/inductive step to make the connection to mathematical induction clear.
     (b) Our inductive step (which I may sometimes call the *recursive case*, and what, in the context of loop invariants, is called the *maintainance* step) will show that if the invariant is true at the end of the last iteration, it'll be true at the end of the next iteration.
  3. We then must show that if the loop invariant is true, our algorithm returns the correct answer after the last iteration. This is sometimes referred to as the *termination* step.

- For Insertion Sort, our loop invariant comes from our observation that after the iteration of the outer loop where $i = k$, the $A[1...i]$ is sorted.

- We can concoct a general structure for our proof by observing the following facts:

1. Proving our base case is simple: before the loop starts (say, with $i = 1$, since our loop starts with $i = 2$), $A[i]$ consists of a single element, and is thus trivially sorted (trivial in the sense that a single item is always sorted!). All of the rest of our work will involve the inductive step.
2. At the end of each iteration, we know the while loop has terminated. This means that one of the two termination conditions was reached (i.e., the condition for continuing the loop was false).

   That is, either $j = 1$, or $A[j-1] \leq A[j]$.

   If $j = 1$, we don't get much useful info... we still need to prove $A[1...i]$ will be sorted by the end of the loop. However, we can restate this as $A[j..i]$, since $j = 1$, which will be helpful later.

   If $A[j-1] \leq A[j]$, we have a useful inequality to use later!
3. Third, we can observe that at the end of each run of our while loop, j is decremented. Before that, $A[j]$ and $A[j-1]$ have been swapped, so looking at the value of j after the loop ends, we change the positions of elements $A[j]$ and $A[j+1]$. In fact, since our loop is decrementing, the only elements that could have been modified within the while loop are elements with index $\geq j$!. This means that $A[1...j-1]$ are *untouched* from the beginning of the loop. Since we assume that $A[1...k]$ is sorted at the beginning of the loop where $i = k+1$ (this is our *inductive hypothesis!*), if $A[1...j-1]$ is untouched, then they should remain sorted after the loop!

- Now, let's think a bit about what it means for an array to be sorted. We will say that $A[1...N]$ is sorted when

$$A[1] \leq A[2] \leq \cdots \leq A[N]$$

And thus for some $1 \leq j \leq N$, we can write this as

$$A[1] \leq A[2] \leq \ldots A[j-1] \leq A[j] \leq \cdots \leq A[N]$$

From Fact 3, we can observe that at the end of the ith iteration $A[1...j-1]$ must be sorted! That is

$$A[1] \leq A[2] \leq \cdots \leq A[j-1]$$

From Fact 2, if we ended up in the case where $A[j-1] \leq A[j]$, we can go further and tack this inequality onto the chain!

$$A[1] \leq A[2] \leq \cdots \leq A[j-1] \leq A[j]$$

This is great! This is a good chunk of our loop invariant! What's remaining is showing that $A[j..i]$ is also sorted. That is, we need to show

$$A[j] \leq \cdots \leq A[i]$$

These are specifically the indices that our while loop has been swapping! So what's left to prove is that our while loop does what it says it does: put the element that starts at index i into it's correct position without breaking the sorted order.

- With the structure of the larger proof coming together, it seems like we'll need to construct and prove a *lemma* (a smaller claim that will help us prove the correctness of our algorithm) along the way. Since the while loop is another loop, we might also want to prove this lemma by induction!
- Now, let's present the full proof in it's formal glory. Though this will hide some of the intuitions that let us get to this final proof structure, this format will make it more straightforward to verify the correctness of the statements involved! Pay careful attention to how the lemma is proven to see another example of a proof by induction.

2.1 Proof of Correctness: Insertion Sort

Lemma *(Swaps preserve order)*: After each iteration of the while loop, $A[j \dots i]$ is sorted. That is,

$$A[j] \leq A[j+1] \leq \dots A[i]$$

Proof: We'll show this by (bounded) induction, for iterations where $1 \leq j \leq i$.

Base Case: Assume $j = i$ (as j is initialized in the algorithm). $A[j \dots i]$ is simply $A[i]$, and, again, an array of length 1 is always sorted!

Inductive Step: Assume that after the iteration where $j = k$ at the end, $A[k \dots i]$ is sorted. We must show that after the iteration where $j = k - 1$ (i.e., after 1 more step!), $A[k-1 \dots i]$ is sorted.[1]

Now, we start reasoning about our code. If we begin a loop with $j = k - 1$, three things can happen:

1. We break out of the loop because $j = k = 1$. [2] Remember that we decrement j as the last step of our loop, and we measure our loop iterations at the end of each loop, so $j = k$ for the condition checking!

   If this is true, then we're at the bound of our bounded induction! We now know, by our inductive assumption, that $A[1 \dots i]$ is sorted, and we need to induct no further!

2. We break out of the loop because $A[k] \not< A[k-1]$.

   This is equivalent to saying $A[k-1] \leq A[k]$. By our inductive assumption, we know

$$A[k] \leq \cdots \leq A[i]$$

[1] Note that we're working backwards here, counting down from i toward 1. This lets us follow the indices of the loop directly!

[2] I've been saying that breaking out of the loop because $j \not> 1$ means $j = 1$. Strictly speaking, $j \not> i$ is equivalent to $j \leq 1$. why can't we break out because $j < 1$? (think about how j gets smaller each loop!)

and so tacking on another inequality to the front gives us

$$A[k-1] \leq A[k] \leq \dots A[i]$$

which is equivalent to saying A[k-1...i] is sorted. This is what we sought to prove, so we're done in this case!

3. We don't break out of the loop and continue for another iteration. This tells us that $A[k-1] > A[k]$ (otherwise, we would have been in case 2!). However, the first line of our while loop swaps the positions of $A[k]$ and $A[k-1]$. Thus, since $A[k-1] > A[k]$ before the swap, $A[k-1] < A[k]$ after the swap! For the sake of our definition of sorting, we can weaken this inequality: If $A[k-1] < A[k]$, $A[k-1] \leq A[k]$. From here, we proceed like in case 2: By our inductive assumption, we know

$$A[k] \leq \dots \leq A[i]$$

and we just saw that $A[k-1] \leq A[k]$, so tack on the new inequality to the chain and

$$A[k-1] \leq A[k] \leq \dots \leq A[i]$$

which is equivalent to saying $A[k-1 \dots i]$ is sorted, as we set out to prove.

It's also worth observing that the other line of the while loop will decrement j, so $j = k-1$ now, so this is the iteration we intended (i.e., the iteration where we end with $j = k-1$ has $A[k-1 \dots i]$ sorted.

Thus, by this bounded induction argument, we can conclude that when the while loop ends with some $1 \leq j \leq i$, we know $A[j \dots i]$ is sorted!

Now we move to the main proof:

Statement *(Loop invariant for InsertionSort)*: After iteration i of insertion sort, $A[1 \dots i]$ is sorted. That is,

$$A[1] \leq A[2] \leq \dots \leq A[i]$$

Base Case: Before the first iteration of the loop, $i = 1$. Observe that $A[1 \dots 1] = A[1]$ and again our (sub)array is trivially sorted, since it's an array with 1 item in it, so we're done.

Inductive Step: Assume that after the iteration where $i = k$, $A[1 \dots k]$ is sorted. Again, this means

$$A[1] \leq A[2] \leq \dots \leq A[k]$$

We must show that after the iteration where $i = k+1$, $A[1 \dots k+1]$ is sorted (I won't expand this one out!).

Now, we know that at the end of this iteration, the while loop has stopped. As we saw in the lemma's proof, we know this means either:

1. $j = 1$, which means, by our lemma, $A[1 \dots i]$ is sorted. Since this is the iteration where $i = k+1$, this tells us $A[1 \dots k+1]$ is sorted, which is our goal. We're done!

2. $A[j] \not< A[j-1]$, which is equivalent to saying $A[j-1] \leq A[j]$.

   Now we make one more observation: We know $j > 1$, and we know that, after each iteration of the loop, the only changes we make to our array are the swaps, which only occur for positions j or greater (where here j refers to the value of j *after the while loop terminates*). This means that $A[1 \dots j-1]$ are unchanged from the beginning of our loop. But, by our inductive assumption, $A[1 \dots k]$ are sorted, and $j \leq i = k+1$, $j - 1 \leq k$. All this means is that $A[1 \dots j-1]$ is unchanged from the beginning of the loop and was a contiguous subarray of the portion of the array that, by our inductive assumption, began the iteration *sorted*! We can then conclude that $A[1 \dots j-1]$ must be sorted at the end of the iteration. Now let's piece everything we know together.

   We just showed that since this part of the array is unaffected by the while loop,

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

   and since we're in the case where we exited the while loop because $A[j] \not< A[j-1]$, we know

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

   And by our lemma (which, as a reminder, uses our inductive assumption!), we know that since we finished an iteration of the while loop,

$$A[j] \leq \cdots \leq A[k+1]$$

   If we stick these all together we get...

$$A[1] \leq \cdots \leq A[j-1] \leq A[j] \leq \cdots \leq A[k+1]$$
$$A[1] \leq \cdots \leq A[K=1]$$

   which is equivalent to saying $A[1 \dots k+1]$. Since this is what we set out to prove, we're done!

   So, we're shown that in all cases, assuming $A[1 \dots k]$ at the beginning of the iteration where $i = k+1$ means that $A[1 \dots k+1]$ is sorted at the end. Combined with our base case, have proven our loop invariant.

   All that's left to say is that because of our loop invariant, after the Nth iteration, we know $A[1 \dots N]$ is sorted. That's the entire array (we assume N is the length of A), so we can conclude that insertion sort works!

2.2 Some Comments (on writing these kinds of proofs)

- Keep in mind that the proof I showed you is meant to a both a model proof as well as a teaching example. That means I try to be much more detailed than I would otherwise be (and much more detailed than I expect the proofs you write to be).
- As far as evaluation of proofs goes, the part that's most important to me is your understanding of the proof technique.
  - Did you choose a reasonable loop invariant?
  - Is the base case you choose reasonable?

  - Do you understand how an inductive step works (the inductive assumption, what you need to prove, etc.)?
  - Are you reasoning about the execution of pseudocode appropriately (you know branching logic sets up different cases to prove, you make sensible judgements about what properties certain operations preserve or change)?
  - Can you formalize claims about code into mathematical statements you can manipulate (i.e., knowing that a list being sorted means $A[1] \leq \cdots \leq A[N]$)?

- What I care the *least* about (but still care a little about!) is you're ability to perform algebraic manipulations on the fly. You'll see a handful of tricks in examples (they show up a lot more in Big-Oh proofs than correctness!), but what's much more important is internalizing the *structure* of the arguments we make. You'll see later on that certain proof techniques lend themselves to algorithms constructed with certain design principles.

- Also of note is the fact that in any timed situation, the things I ask you to prove will be much simpler than this proof of insertion sort.

- A small secret is the fact that proofs I ask you to write will often look similar to proofs I show you in lecture. In the rare cases that the technique is something new, I'll give you plenty of time to work through it in a homework assignment, using techniques similar to those you've seen in an example.

3 Induction for Recursive Algorithms

- Luckily, mathematical induction lends itself much better to proving recursive algorithms correct than iterative algorithms.[3]

- Whether you're writing a recursive function or an inductive proof, you worry about a base case (which, hopefully, is something easy to prove), and then we either write a recursive case or prove an inductive step. The reason I might sometimes refer to the inductive step of a proof as the recursive case is because they are pretty much the same thing!

- Check out the following proof for Binary Search, which will let us pull out *strong* induction.

3.1 Proof of Correctness: Binary Search

- First, let's specify what correctness means here. A search algorithm takes in an Array A and an target element e and returns an index i such that $A[i] == e$ if $e \in A$, or $NULL$ otherwise.

  - Since this is BinarySearch, we have an additional assumption: It only works if A is sorted! We can't escape talking about sorting, unfortunately.

- This means that we need to prove two things (we love breaking things into cases!)

  1. **If the algorithm returns index** i: We must show $A[i] == e$

[3]This is not unrelated to the fact that functional programming languages are often used in *formal verification*, an area of CS where you build programs that generate proofs that a piece of code is correct according to a set of specifications. If this sounds cool, check out Coq/Gallina

```
function BinarySearch(Array A, Target e)
    if N == 0 then
        return NULL
    end if
    midpoint ← ⌈N/2⌉
    if A[midpoint] == e then
        return midpoint
    else if e < A[midpoint] then
        return BinarySearch(A[1 ... midpoint − 1], e)
    else
        return midpoint + BinarySearch(A[midpoint + 1 ... N], e)
    end if
end function
```

2. **If the algorithm returns** $NULL$: We must show $e \notin A$.

Okay — Let's prove correctness by induction directly(no loop invariants or any other tricks!). First, let's establish that we are going to use strong induction over the size of the Array, N. We do this because we realize that the recursive calls are always on smaller arrays (but not arrays of length $N-1$, so weak induction is not enough).

Base Case: We are going to prove that this algorithm returns the correct answer for arrays of size 0.

Again, this is easy, but strange: An empty array cannot contain e, so we should always return $NULL$. This happens, because if $N = 0$, we immediately enter the the first conditional and return $NULL$. We're done!

Inductive Step/Recursive Case: Now things get a little messy, because we have a lot of cases. These cases, thankfully, can be determined relatively easily: They're the cases of the if-elif-else block!

First, let's be clear about what our inductive assumption is: BinarySearch(A', e) will return the correct answer as long as A' is sorted and the length of A, N', is less than N. Note that this is by *strong* induction, so this is true for all $N' < N$, and we must show this holds for N.

So either

1. $A[midpoint] == e$. Here, we immediately return $midpoint$. What we need to show is exactly what we assume in this case: $A[midpoint] == e$, so we're done!

2. $e < A[midpoint]$. Here, we have a few things to show. We enter the **else if** case, and immediately return BinarySearch($A[1 \dots midpoint - 1]$, e).

   First, note that $midpoint - 1 < N$, and thus the length of the array argument is smaller than N. Then note that $A[1 \dots midpoint - 1]$ is sorted, because it's just a contiguous subarray of A, which is sorted by assumption. We can write this out in more detail by observing that A being sorted means

   $$A[1] \leq \cdots \leq A[midpoint - 1] \leq \cdots \leq A[N]$$

   which implies

$$A[1] \leq \cdots \leq A[midpoint - 1]$$

which is the same as saying $A[1 \ldots midpoint-1]$ is sorted. This means that the array argument to this recursive call has length $< N$ and is sorted, which means by our inductive assumption, means that the answer returned is correct! But here's where things get annoying (but not *tricky*), because there are two cases :

(a) The call **returns** $NULL$: Since the recursive call is guaranteed to be correct, this means that $e \notin A[1 \ldots midpoint - 1]$. Since this call also returns $NULL$, we need to show that $e \notin A[midpoint \ldots N]$.

Thankfully we have some tools. Because we're in case 2, we know that $e < A[midpoint]$. Further, we know that A is sorted, so

$$A[midpoint] \leq A[midpoint + 1] \leq \cdots \leq A[N]$$

If $e < A[midpoint] \leq \ldots A[N]$, we know $e < A[k]$ for all $midpoint \leq k \leq N$, which means that $e \neq A[k]$ for all $midpoint \leq k \leq N$ (strict inequalities are nice!). This implies that $e \notin A[midpoint \ldots N]$, and with our inductive assumption from earlier, we can conclude $e \notin A[1 \ldots N]$, and confirm that returning $NULL$ was the right call!

(b) The call returns some index i. This means that for $B = A[1 \ldots midpoint]$, $B[i] == e$ by our strong inductive assumption (a smaller call is always correct!).

Of course, since the ith element of B is the ith element of A (we just swapped notation to B to avoid messy notation!), this implies that $A[i] == e$, which is exactly what we need to show.

And that covers all subcases of case 2.

3. $A[midpoint] \neq e$ and $e \not< A[midpoint]$: Note that in the **else** case, you always know the other conditions are false, since you would have been in that other case if they were true! First, we can see that the two facts we know can combine to show $e > A[midpoint]$. If we're not smaller or equal, we're greater!

   From here on out, this case should look like a perfect mirror of case 2.

   By our inductive assumption, we know our call to BinarySearch returns the right answer, so two cases:

   (a) The call returns $NULL$: We thus know $e \notin A[midpoint + 1 \ldots N]$. We also know in this case that $e > A[midpoint]$, and since A is sorted, we know

   $$A[1] \leq \cdots \leq A[midpoint]$$

   And we can thus conclude that

   $$e > A[midpoint] \geq A[midpoint - 1] \geq \cdots \geq A[1]$$

   And thus $e \notin A[1 \ldots midpoint]$. Thus $e \notin A$, and we return $NULL$ correctly.[4]

[4] To be entirely clear, we assume $midpoint + NULL$ is $NULL$. One of the nice parts of writing in pseudocode is that we can trust a reader's intuition to know that returning a number plus $NULL$ should return $NULL$. Depending on your programming language, a compiler may not like something like this.

(b) The call returns some index i: We now know for $B = A[midpoint{+}1 \ldots N]$, $B[i] == e$. B is simply A with the first $midpoint$ elements removed, so we know $B[i] == A[midpoint + i]$, and since the call is guaranteed to be correct by strong induction, we know

$$B[i] == A[midpoint + i] == e$$

The second half of this equality is exactly what we need to show in order to demonstrate, since we return $midpoint + i$. We're done with this case!

Now we've handled all 3 cases of the inductive step (which we can verify because those are the only places where we return a value), and can conclude, by strong induction, that for sorted arrays of length $N \geq 0$, BinarySearch is correct!

4 A Closing Note

The most difficult thing about proving the correctness of algorithms is turning your intuitions about how the algorithm works into the formal pieces necessary for a certain proof technique. I always recommend beginning by informally convincing yourself that the algorithm works, and inspecting why you think that is.

- If you can write working code (which you can — you can't pass COMP128 without!), you have intuitions about how and why code works.
- You also passed Discrete Math, which means that you're comfortable with the kinds of mathematical reasoning we'll be using (mainly induction!).
- You just need to bridge the gap between those two things. The *just* in that sentence should not mislead you though — this is **hard**. It's perfectly normal for things to seem a little overwhelming at first, but the way to overcome that thoughtful practice and a willingness to challenge yourself!