Proofs of Correctness: Recursive Algorithms

COMP 221 — Suhas Arehalli

- Luckily, mathematical induction lends itself much better to proving recursive algorithms correct than iterative algorithms.[1]
- 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.

1 Proof of Correctness: Binary Search

```
function BinarySearch(Array $A$, Target $e$)
    if N == 0 then
        return $NULL$
    end if
    $midpoint \leftarrow \lceil \frac{N}{2} \rceil$
    if $A[midpoint] == e$ then
        return $midpoint$
    else if $e < A[midpoint]$ then
        return BinarySearch($A[1 \dots midpoint - 1]$, $e$)
    else
        return $midpoint$ + BinarySearch($A[midpoint + 1 \dots N]$, $e$)
    end if
end function
```

- 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] 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

1. **If the algorithm returns index** i: We must show $A[i] == e$
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($A[1 \ldots midpoint-1]$, e).

   First, note that $midpoint - 1 < N$, and thus the length of the array argument is smaller than N. Then note that $A[1 \ldots midpoint-1]$ 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.[2]

   (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!

[2]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.