CS231 Algorithms
Prof. Lyn Turbak
Wellesley College

Handout # 2
February 11, 2022
Revised February 20, 2022

Loop Invariants and Mutable Array Sorting

- [2022/02/20 7:30pm] Modulo bugs that might be found and may need to be fixed, this is the final version of the Loop Invariant handout.

Proving Correctness

We want to be able to prove that an algorithm is *correct* – i.e., that it satisfies the specification of a solution for the problem being solved.

The divide/conquer/glue (DCG) problem solving strategy naturally leads to algorithms involving recursive functions. Proving such functions correct can be done with induction — something we'll focus on in the next lecture.

But many algorithms involve loops. How do we prove loop-based algorithms correct? A proof technique for this common situation is called **loop invariants**.

Loop Invariants

The **loop invariants** proof technique is a specialization of proof-by-induction for iterations (as in loops or tail-recursive functions). Given a loop with state variables $s_1, \ldots, s_k$, the technique involves the following steps. We'll use the notation $s_j(i)$ to denote the value of state variable s_j at the beginning of the ith iteration of the loop. We'll assume the iterations are 1-index — i.e., for the first iteration, $i = 1$ (not 0).

1. **Specify one or more *invariants* that hold among the state variables of the loop**. Formally, this is a k-ary relation $LI(s_1, \ldots, s_k)$.

2. **Show that the loop invariants hold the first time the loop is entered**. I.e., it is necessary to show that $LI(s_1(1), \ldots, s_k(1))$ holds.

3. **Show that if the loop invariants hold at the beginning of iteration i, then they hold at the beginning iteration $i + 1$**, i.e., after the body of the loop has executed, regardless of the path taken through the body. Formally, it must be shown that for all i, $LI(s_1(i), \ldots, s_k(i))$ implies $LI(s_1(i + 1), \ldots, s_k(i + 1))$.

4. **Show that that the loop terminates.** This is usually done by defining **metric** function $M(s_1(i), \ldots, s_k(i))$ returning a nonnegative integer that characterizes the size of the problem at iteration i, and showing that the metric strictly decreases at every iteration. Since the function returns a nonnegative integer and decreases at every iteration, it must eventually reach 0, at which point the loop terminates.

5. **Show that the terminating state satisfies the desired correctness properties.** The terminating state corresponds to the beginning of an iteration that is not executed. Formally, if the loop terminates when condition at the beginning of the tth iteration is tested, it must be shown that $LI(s_1(t), \ldots, s_k(t))$ implies the desired correctness property.

Example: while-loop based Factorial Program

Here we illustrate the loop invariant technique in the context of a **while** loop based version of a factorial function. (This is the same as Example 3 in the Appendix to PS1.)

```
function FactWhile(n)                    ▷ Return n!, calculating it via a while loop
    prod ← 1
    num ← n
    while num > 0 do
        prod ← num · prod
        num ← num − 1
    return prod
```

Below is an iteration table for the **while** loop in the invocation FactWhile(5). The table has columns indexed by state variables that define the state of the execution of the loop. We assume there is an implicit index variable i that counts the iterations of the loop, i.e., the number of times the condition in the **while** loop is tested. We will refer to this implicit variable in our analysis. The rows of the table show the values of the state variables at the **beginning** of the iteration i — i.e., before the loop test is performed and and body of the loop is (potentially) executed.

i	num	$prod$
1	5	1
2	4	5
3	3	20
4	2	60
5	1	120
6	0	120

The state variables num and $prod$ are effectively functions of the implicit index variable i. We write num_i and $prod_i$ for the values of these variables at iteration i. E.g., in the above table, $num_4 = 2$ and $prod_4 = 60$.

Next we show the five parts of a general loop invariant proof in the context of this example.

FactWhile: Specify the Loop Invariants

In this example, there is one loop invariant (but in general, there might be several).

(FactWhileLI) $num_i! \cdot prod_i = n!$ (where n is the parameter of FACTWHILE).

To intuitively understand this invariant, let's verify it in each row of the iteration table:

i	num	$prod$	$num_i! \cdot prod_i$
1	5	1	$5! \cdot 1 = 120 = 5!$
2	4	5	$4! \cdot 5 = 120 = 5!$
3	3	20	$3! \cdot 20 = 120 = 5!$
4	2	60	$2! \cdot 60 = 120 = 5!$
5	1	120	$1! \cdot 120 = 120 = 5!$
6	0	120	$0! \cdot 120 = 120 = 5!$

You can think of it this way: when the iteration begins, all information about what we wish to compute is contained in $num_1 = 5$ (i.e., we want to calculate 5!) and no information is contained in $prod_1 = 1$ (the identify element for multiplication). As the iteration proceeds, it "transfers" information from num to $prod$ so that, at the end, no information is in $num_6 = 1$ (the identity element for multiplication) and all information is in $prod_6 = 120 = 5!$. From this perspective, the goal of the loop is to transfer information from some state variables to others in such a way that the desired answer is in one or more state variables at the end of the loop.

FactWhile: Show that the Invariants Hold on Entry to Loop

Proof: Our one invariant holds before the first loop iteration is executed:

$$num_1! \cdot prod_1 = n! \cdot 1 = n! \heartsuit$$

FactWhile: Show that Each Loop Iteration Preserves the Invariants

In other words, we need to show that if the invariant holds at the beginning of iteration i (before the conditional test is performed and the body is potentially executed), then it holds at the end of the iteration (after the body has been executed), which is the same as saying it holds at the beginning of iteration $i + 1$.

Proof In the case of FACTWHILE, assume that $num_i! \cdot prod_i = n!$. (Think of this as being similar to the inductive hypothesis (IH) in a proof by induction.) We want to show that $num_{i+1}! \cdot prod_{i+1} = n!$. From the definition of FACTWHILE, we have the following facts:

(Update *num*) $num_{i+1} = num_i - 1$

(Update *prod*) $prod_{i+1} = num_i \cdot prod_i$

We can combine these facts to yield the desired proof:

	$num_{i+1}! \cdot prod_{i+1}$	
=	$(num_i - 1)! \cdot (num_i \cdot prod_i)$	by (Update *num*) and (Update *prod*)
=	$num_i! \cdot prod_i$	by the definition of factorial
=	$n!$	by the assumption that (FactWhileLI) holds for iteration i $\heartsuit$

FactWhile: Show Termination

When a loop uses a **while** loop as opposed to a **for** loop, we need to prove that it will eventually terminate, because it may not be apparent that the the number of iterations can be bounded in advance.

To show termination, we often define a metric function M on all the state variables, and show that the value of this function decreases on every iteration until some end point.

Proof

In the case of FactWhile, we define $M(num_i, prod_i) = num_i$. Since $num_{i+1} = num_i - 1$, M clearly decreases by 1 with each iteration. Assuming that n is nonnegative, num_1 is nonnegative, and by the definition of FactWhile, *num* never goes below 0. Therefore, the **while** loop in FactWhile terminates at the beginning of iteration $n + 1$ (e.g., $i = 6$ in the iteration tables shown above). ♡

FactWhile: Show Desired Properties

We wish to show that FactWhile(n) returns $n!$.

Proof

When FactWhile terminates in the $(n + 1)$th iteration, $num_{n+1} = 0$. So:

$$\begin{array}{rcll} prod_{n+1} & = & 0! \cdot prod_{n+1} & \text{because } 0! = 1 \\ & = & num_{n+1}! \cdot prod_{n+1} & \text{because } num_{n+1} = 0 \\ & = & n! & \text{by the loop invariant (FactWhileLI)} \end{array}$$

Therefore, the result returned by FactWhile(n), which is the final value of the state variable *prod*, is indeed $n!$. ♡

Proving Correctness of Sorting Algorithms on Mutable Arrays

Loop invariants are commonly used to reason about algorithms involving mutable arrays. We will now consider how to prove the correctness of sorting algorithms on mutable arrays.

We'll assume 1-indexed mutable fixed-length arrays that hold simple values like numbers, characters, or strings, but we could easily adapt the values to be objects with a key used for sorting.

Given a length-n array whose initial state is A_{init}, the goal of a sorting algorithm is to change the state of the array to A_{final} such that:

1. The elements of A_{final} are the same as A_{init}, modulo their ordering.
2. The elements of A_{final} are sorted — i.e., for all $i \in [1..n-1]$, $A[i] \leq A[i+1]$

Example: Insertion Sort on Mutable Arrays

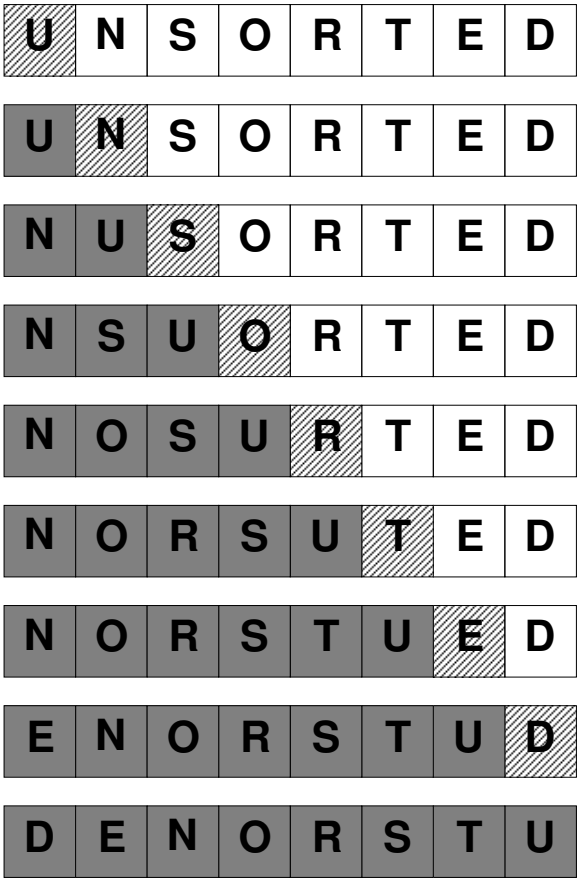
Loop invariants are commonly used to reason about algorithms involving mutable arrays. Here we apply the loop invariant technique to insertion sort, an easy-to-understand sorting algorithm:

```
function InsertionSort(A)                          ▷ Sort the elements of 1-indexed mutable array A
    for all h ∈ [2..len(A)] do
        Insert(A, h)

function Insert(A, k)                              ▷ Assume A[1..(k − 1)] is sorted and k ≤ len(A)
                                                   ▷ Modify A so that A[1..k] has the same elements as when entered, but is sorted.
    j ← k
    while j ≥ 2 and A[(j − 1)] > A[j] do
        Swap(A, j − 1, j)
        j ← j − 1

function Swap(A, p, q)                             ▷ Swap the contents of slots indexed p and q within array A
    A[p], A[q] ← A[q], A[p]                        ▷ Parallel assignment to express the swap.
```

Depiction of Insertion Sort



Swap-based Sorting Algorithms Preserve Elements

InsertionSort is an example of a **swap-based** sorting algorithm, in which all changes to the array are made by swapping elements at two given indices.

Swap-based algorithms obviously preserve the elements in an array, thereby satisfying the first correctness property of an array sorting algorithm: "The elements of A_{final} are the same as A_{init}, modulo their ordering."

This property needs more attention when other approaches are used to reorder the array elements.

Game Plan for Proving Correctness of Sorting for InsertionSort

We just need to focus on sorting correctness property 2: "The elements of A_{final} are sorted — i.e., for all $i \in [1..n-1]$, $A[i] \leq A[i+1]$"

There are two loops in InsertionSort: one in Insert and one in InsertionSort itself. We'll use loop invariants on both loops in order to prove the result of InsertionSort is sorted.

Insert: Specify the Loop Invariants

Let A_i denote the contents of array A at the beginning of the ith iteration of the **while** loop (where i starts at 1) and j_i denote the value of state variable j at the beginning of the ith iteration.

Here are **three**[1] loop invariants for Insert (all of which must hold at every iteration)

InsertLI1 $A_i[1..(j_i - 1)] = A_1[1..(j_i - 1)]$
(and so $A_i[1..(j_1 - 1)]$ is sorted, because $A_1[1..(k-1)]$ is assumed to be sorted). Note that two array **segments** are considered to be equal iff (1) they have the same number of elements and (2) the elements at the corresponding locations are equal.

InsertLI2 $A_i[(j_i + 1)..k] = A_1[j_i..(k-1)]$
(and so $A_i[(j_1 + 1)..k]$ is sorted, because $A_1[1..(k-1)]$ is assumed to be sorted)

InsertLI3 if $j_i < k$, then $A_i[j_i] < A_i[j_i + 1]$

[1] I originally presented **two** invariants, but for reasons to be explained, those were insufficient and you really need these three.

Insert: What Do We Know?

When doing loop invariant proofs (or really any proofs) it's a good idea to carefully write down in one place all the facts that you know that will be important for the proof. It's also a good idea to give each fact a name that you can refer to in the justification steps of the proof. If you don't use one of the named facts at some point in your proof, this is a strong indication that your proof is missing something!

In the case of the loop invariant proof for the correctness of INSERT, here's what we know:

From the INSERT *function*:

(Init j) $j_1 = k$

(Update j) $j_{i+1} = j_i - 1$

(Swap1) $A_{i+1}[j_i] = A_i[j_{i-1}]$ (if $A_i[j_{i-1}] > A_i[j_i]$))

(Swap2) $A_{i+1}[j_{i-1}] = A_i[j_i]$ (if $A_i[j_{i-1}] > A_i[j_i]$))

(NoSwap) $A_{i+1}[h] = A_i[h]$ (if $h \notin \{j_{i-1}, j_i\}$). I.e, elements at unswapped indices remain the same.

From the loop invariants:

(InsertLI1 Assumption) $A_i[1..(j_i - 1)] = A_1[1..(j_i - 1)]$

(InsertLI2 Assumption) $A_i[(j_i + 1)..k] = A_1[j_i..(k - 1)]$

(InsertLI3 Assumption) if $j_i < k$, then $A_i[j_i] < A_i[j_i + 1]$

In the above facts, indices have been highlighted in a different color to emphasize them. Being careful with indices in a loop invariant proof is a key element of success!

Now that we have all the facts, the idea is to show each step of a loop invariant proof by using the facts.

Insert: Show that the Invariants Hold on Entry to Loop

It's good practice to begin a proof by (1) clearly stating all the facts you know (done above) and (2) clearly stating what you want to prove.

Here we want to prove each of the three loop invariants holds when $i = 1$, so we substitute i for 1 in the equality defining each loop invariant:

InsertLI1: Want to prove $A_1[1..(j_1 - 1)] = A_1[1..(j_1 - 1)]$

InsertLI1: Proof

This proof is trivial, since the desired equality is clearly true when 1 is substituted for i. ♡

InsertLI2: Want to prove $A_1[(j_1 + 1)..k] = A_1[j_1..(k - 1)]$

InsertLI2: Proof

$A_1[(j_1 + 1)..k]$	=	$A_1[(k + 1)..k]$	(Init j)
	=	an empty array segment	because $(k + 1) > k$ in $A_1[(k + 1)..k]$
	=	$A_1[k..(k - 1)]$	because $k > (k - 1)$ in $A_1[k..(k - 1)]$
	=	$A_1[j_1..(k - 1)]$	(Init j) ♡

InsertLI3: Want to prove if $j_i < k$, then $A_i[j_i] < A_i[j_i + 1]$

InsertLI3: Proof

$j_i = k$ (by (Init j)) and $k \not< k$, so the "if" test in "if $j_i < k$, then ..." is false. By the rules of logic, in the implication expression"if p then q" (also written $p \implies q$), if the antecedent p is false, the implication expression itself is considered true (and is said to be "vacuously true" in this case). For example, the logical expression "if it's raining I use an umbrella" is vacuously true when it's not raining, whether or not I am using an umbrella. So InsertLI3 is true when $i = 1$. ♡

Insert: Show that Each Loop Iteration Preserves the Invariants

Here we want to prove for each of the three loop invariants that if it holds at the beginning of iteration i, if the loop body is executed, then it holds at the beginning of iteration $i+1$ (which is the same as saying it holds at the end of iteration i).

The facts (InsertLI1 Assumption), (InsertLI2 Assumption), and (InsertLI3 Assumption) will play an important role in these proofs:

InsertLI1: Want to prove $A_{i+1}[1..(j_{i+1}-1)] = A_1[1..(j_{i+1}-1)]$

InsertLI1: Proof

$A_{i+1}[1..(j_{i+1}-1)]$	$=$	$A_{i+1}[1..((j_i-1)-1)]$	(Update j)
	$=$	$A_{i+1}[1..(j_i-2)]$	by arithmetic
	$=$	$A_i[1..(j_i-2)]$	by (NoSwap) at each index in $[1..(j_i\text{-}2)]$
	$=$	$A_1[1..(j_i-2)]$	(InsertLI1 Assumption) on all but last element of $A_i[1..(j_i-1)]$
	$=$	$A_1[1..((j_i-1)-1)]$	by arithmetic
	$=$	$A_1[1..(j_{i+1}-1)]$	(Update j) ♡

InsertLI2: Want to prove $A_{i+1}[(j_{i+1}+1)..k] = A_1[j_{i+1}..(k-1)]$

For this proof we'll introduce an operator $\oplus$ for concatenating two contiguous array segments in a given array A: $A[p..q] \oplus A[(q+1)..r]$ denotes the array segment $A[p..r]$.

InsertLI2: Proof

$A_{i+1}[(j_{i+1}+1)..k]$	$=$	$A_{i+1}[((j_i-1)+1)..k]$	(Update j)
	$=$	$A_{i+1}[j_i..k]$	by arithmetic
	$=$	$A_{i+1}[j_i] \oplus A_{i+1}[(j_i+1)..k]$	by definition of $\oplus$
	$=$	$A_i[j_i-1] \oplus A_{i+1}[(j_i+1)..k]$	(Swap1)
	$=$	$A_i[j_i-1] \oplus A_i[(j_i+1)..k]$	by (NoSwap) at each index in $[(j_i\text{+}1)$ .. k]
	$=$	$A_i[j_i-1] \oplus A_1[j_i..(k-1)]$	(InsertLI2 Assumption)
	$=$	$A_1[j_i-1] \oplus A_1[j_i..(k-1)]$	(InsertLI1 Assumption) at $A_i[j_i\text{-}1]$
	$=$	$A_1[(j_i-1)..(k-1)]$	by definition of $\oplus$
	$=$	$A_1[j_{i+1}..(k-1)]$	(Update j) ♡

InsertLI3: Want to prove if $j_{i+1} < k$, then $A_{i+1}[j_{i+1}] < A_{i+1}[j_{i+1}+1]$

InsertLI3: Proof

For $i \geq 1$, if iteration i is reached and the loop is executed, $j_{i+1} < k$ (by (Update j)), so we need to show $A_{i+1}[j_{i+1}] < A_{i+1}[j_{i+1}+1]$ for all such i.

$A_{i+1}[j_{i+1}]$	$=$	$A_{i+1}[j_i-1]$	(Update j)
	$=$	$A_i[j_i]$	(Swap2)
	$<$	$A_i[j_i-1]$	by (Swap2) precondition
	$=$	$A_{i+1}[j_i]$	(Swap 1)
	$=$	$A_{i+1}[(j_i-1)+1]$	arithmetic
	$=$	$A_{i+1}[j_{i+1}+1]$	(Update j) ♡

Insert: Show Termination

We need to show that the **while** loop in INSERT cannot run infinitely — i.e., it must terminate in a finite number of iterations.

Proof

In INSERT, the **while** loop continuation condition is $j \geq 2$ and $A[(j-1)] > A[j]$. We know from (Init j) that j is intialized to k (i.e., $j_1 = k$) where k is assumed to be an integer. There are two cases:

Case 1: $k < 2$ In this case, the **while** loop continuation condition is initially false because $j_1 \not\geq 2$. So the loop body is never executed in this case, and the loop terminates.

Case 2: $k \geq 2$ In this case, by (Update j), we know the value of j is decremented by 1 on each iteration of the loop, and so j must reach a value < 2 in a finite number of steps, terminating the loop. The loop can terminate even earlier in the case where $A_i[j_i - 1] \leq A_i[j_i]$ at the beginning of some iteration i. ♡

Insert: Show Desired Properties

The key property of INSERT that we wish to show is that when the **while** loop terminates at final iteration f (i.e., the iteration for which the **while** loop continuation condition becomes false), the array segment $A_f[1..k]$ is sorted.

Proof

There are two cases, based on the reason why the loop terminated:

Case 1: $j_f < 2$**:** In this case, there are two subcases:

Subcase 1: $k < 2$: In this subcase, the **while** loop continuation condition is initially false because $j_1 \not\geq 2$. So the loop body is never executed in this case, and the initial array A is unchanged. But since $k < 2$, A is either a singleton array or an empty array, both of which are sorted, so we verify $A_f[1..k]$ is sorted in this subcase.

Subcase 2: $k \geq 2$**:** In this subcase, the **while** loop continuation condition was true at least once. and the loop terminates when the value initially at $A_1[k]$ has bubbled down to the position $A_f[1]$. In this subcase $j_f = 1$ and we know that:

1. $A_f[2..k] = A_f[(j_f + 1)..k]$ by arithmetic
   $= A_1[j_f..(k-1)]$ by (InsertLI2 Assumption)
   $= A_1[1..(k-1)]$ because $j_f = 1$ in this case
2. $A_f[1] = A_f[j_f]$ because $j_f = 1$ in this case
   $< A_f[j_f + 1]$ by (InsertLI3 Assumption) and the fact that $j_f < k$
   $= A_f[2]$ because $j_f = 1$ in this case

By the preconditions on INSERT, $A_1[1..(k-1)]$ is assumed to be sorted. From (1), we deduce that $A_f[2..k]$ is sorted. Then from (2), we can deduce that $A_f[1..k]$ is sorted.

Case 2: $j_f \geq 2$ **and** $A_f[j_f - 1] \leq A_f[j_f]$**:** In this case, the loop stops when the element at $A_f[j_f]$ can't bubble down any further. We know that

1. $A_f[1..(j_f - 1)] = A_1[1..(j_f - 1)]$ (InsertLI1 Assumption)
2. $A_f[j_f] < A_f[j_f + 1]$ (InsertLI3 Assumption)
3. $A_f[(j_f + 1)...k] <= A_f[j_f..(k-1)]$ (InsertLI2 Assumption)

Since $A_1[1..(k-1)]$ is assumed to be sorted, $A_f[1..(j_f - 1)]$ is sorted (by (1)) a and $A_f[(j_f + 1)...k]$ is sorted (by (3)). By (2), $A_f[j_f] < A_f[j_f + 1]$, and by the assumption of for this case, $A_f[j_f - 1] \leq A_f[j_f]$, so we conclude that $A_f[1..k]$ is sorted. ♡

Can Weaker Loop Invariants Work for Insert?

The three loop invariants for INSERT seem awfully complicated. Can't they be simplified? For example, here are two simpler loop invariants:

InsertLI1′ $A_i[1..(j_i - 1)]$ is sorted.

InsertLI2′ $A_i[j_i..k]$ is sorted.

If we try to use the loop invariant machinery with the above invariants, we hit a snag. When we swap $A_i[(j_i - 1)]$ and $A_i[j_i]$, we know that $A_i[(j_i - 1)] > A_i[j_i]$, but we don't know that $A_i[(j_i - 1)] \leq A_i[(j_i + 1)]$, and so can't conclude that $A_{i+1}[(j_{i+1}..k]$ is sorted! We need the additional information that $A_i[(j_i - 1)] = A_1[(j_i - 1)]$ and $A_i[(j_i + 1)] = A_1[j_i]$ in conjunction with the fact that $A_1[1..(k - 1)]$ is sorted to make this part of the proof go through.

Loop Invariant and Known Fact for InsertionSort

Now that we know that INSERT is correct, we're ready to prove that INSERTIONSORT works as advertised. This is another loop invariant proof, but this one is much simpler than the loop invariant proof for INSERT.

Recall that INSERTIONSORT is defined as:

```
function InsertionSort(A)                    ▷ Sort the elements of 1-indexed mutable array A
    for all h ∈ [2..len(A)] do
        Insert(A, h)
```

Here is the loop invariant for INSERTIONSORT

(**InsertionSortLI**] At the beginning of iteration i (1-indexed), $A[1..min(i, len(A)]]$ is sorted

Why does the loop invariant use $min(i, len(A))$ rather than just i? As we'll see below, in the special case where A is empty, $A[1..1]$ isn't defined, but $A[1..min(i, len(A))] = A[1..min(i, 0)] = A[1..0]$ **is** defined as the empty array.

And here are the key facts we'll use use in our proof:

(Value of h) $h_i = i + 1$, because h is initialized to 2 and is incremented thereafter.

(Insert Preserves Sortedness) If $A_i[1..i]$ is sorted, and $i + 1 \leq len(A)$, then $A_{i+1}[1..(i + 1)]$ (which is state of the first $i + 1$ slots of A after the call INSERT$(A, i + 1)$) is sorted.

InsertionSort: Show that the Invariants Hold on Entry to Loop

Proof: When $i = 1$, there are two cases:

Case 1: $len(A) = 0$**:** In this case, $A[1..min(i, len(A))] = A[1..min(1, 0)] = A[1..0]$ (an empty array), which is sorted.

Case 2: $len(A) \geq 0$**:** In this case, $A[1..min(i, len(A))] = A[1..min(1, 1)] = A[1..1]$ (an array with 1 element), which is sorted. ♡

InsertionSort: Show that Each Loop Iteration Preserves the Invariants

Here we show that if (InsertionSortLI) holds at the beginning of loop iteration i, it holds at the beginning of iteration $i + 1$, which is the same as saying it holds at the end of iteration i (after the loop body has been executed).

That is, we wish to show that if $A_i[1..min(i, len(A)]$ is sorted, and the body of the loop is executed, then $A_{i+1}[1..min(i + 1, len(A)]$ is sorted.

We also know that for the body of the loop to be executed:

$$\begin{array}{rcll} i & < & i+1 & \text{by arithmetic} \\ & = & h_i & \text{(Value of } h\text{)} \\ & \leq & len(A) & \text{by } \textbf{for all} \text{ loop in body of } \textsc{InsertionSort} \end{array}$$

So we conclude $min(i, len(A)) = i$, and we can simplify the statement of what we wish to show:

if $A_i[1..i]$ is sorted, and the body of the loop is executed, then $A_{i+1}[1..(i + 1)]$ is sorted.

Proof

Assume $A_i[1..i]$ is sorted. If the body of the loop is executed, we know from above that $i + 1 = h_i \leq len(A)$. By (Insert Preserves Sortedness), we can conclude that $A_i[1..(i + 1)]$ is sorted. ♡

InsertionSort: Show Termination

Proof: Because InsertionSort is written in terms of a **for all** loop, it clearly terminates at the end of iteration where $h_i = len(A)$, which is when $i = len(A) - 1$. This is the same as the beginning of the iteration where $i = len(A)$. ♡

InsertionSort: Show Desired Properties

Proof: When the **for all** loop terminates at the beginning of iteration $i = len(A)$, by (InsertionSortLI), $A[1..min(len(A), len(A)] = A[1..len(A)]$ is sorted. ♡

A version of Insert that does not use Swap

We conclude our discussion of InsertionSort by considering an alternative version of Insert that does not use Swap:

function Insert′(A, k) ▷ Assume $A[1..(k-1)]$ is sorted and $k \leq len(A)$
▷ Modify A so that $A[1..k]$ has the same elements as when entered, but is sorted.
$insertVal \leftarrow A[k]$
$j \leftarrow k$
while $j \geq 2$ and $A[(j-1)] > A[j]$ **do**
$A[j] \leftarrow A[j-1]$
$j \leftarrow j-1$
$A[j] \leftarrow insertVal$

From an implementation perspective, Insert′ is preferable to Insert, because it uses fewer assignments to array slots. However, it's a bit more challenging to proof correct. Why? For simplicity, assume that the elements in $A[1..k]$ are distinct. Then after execution of the first iteration of the **while** loop and before the loop terminates, the array segment $A_i[1..k]$ does **not** contain *insertVal* but **does** contain duplicated elements at indices j_i and j_{i+1}. For example, when $A_1[1..5]$ is `N O S U R`, A_2 has a duplicate of `U` and A_3 has a duplicate of `S`.

i	$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$
1	N	O	S	U	R
2	N	O	S	U	U
3	N	O	S	S	U

Only after the **while** loop terminates is the `S` in $A_3[3]$ replaced by $insertVal =$ `R` to yield the result array values `N O R S U`:

So it's not as obvious as with Insert that the the array A after Insert′ returns contains exactly the same elements as the array A when Insert′ is called.

Proving this fact and the sortedness of the array returned by Insert′ requires changing to the loop invariants. InsertL1 and InsertL2 are the same as before, but we need to change **InsertLI3** to **InsertLI3′**:

InsertLI1 $A_i[1..(j_i-1)] = A_1[1..(j_i-1)]$
(and so $A_i[1..(j_1-1)]$ is sorted, because $A_1[1..(k-1)]$ is assumed to be sorted). Note that two array **segments** are considered to be equal iff (1) they have the same number of elements and (2) the elements at the corresponding locations are equal.

InsertLI2 $A_i[(j_i+1)..k] = A_1[j_i..(k-1)]$
(and so $A_i[(j_1+1)..k]$ is sorted, because $A_1[1..(k-1)]$ is assumed to be sorted)

InsertLI3′ if $j_i < k$, then $A_1[k] = insertVal < A_i[j_i+1]$. (Since the value of *insertVal* doesn't change during the loop, we don't need to subscript it.)

We do not give a proof here, but encourage the reader to think about why the modified loop invariant **InsertLI3′** is sufficient to show both that(1) Insert′ preserves the array elements in $A[1..k]$ and (2) Insert′ guarantees that $A[1..k]$ is sorted when it returns.