Loop Invariants

Stefan Muller, based on material by Jim Sasaki

CS 536: Science of Programming, Fall 2023
Lectures 14-15

1 Setup

1.1 Context: Why Do We Care About wp, sp?

Remember proof outlines, like this one:

$$
\begin{array}{ll}
 & \{T\} \Rightarrow \{1 = 1\} \\
x := \overline{1} & \{x = 1\} \\
\textsf{if } (x > 0) \textsf{ then} & \{x = 1 \wedge x > 0\} \Rightarrow \{1 = 1\} \\
\quad z := 1 & \{z = 1\} \\
\textsf{else} & \{x = 1 \wedge x \leq 0\} \Rightarrow \{0 = 1\} \\
\quad z := 0 & \{z = 1\} \\
\textsf{fi} & \{z = 1\}
\end{array}
$$

Do we really need to write out all those annotations? Maybe we can infer some using wp and sp. In fact, we can. Let's try. First, we can remove the preconditions of the branches; it's clear what those need to be from the precondition of the if and the condition.

$$
\begin{array}{ll}
 & \{T\} \Rightarrow \{1 = 1\} \\
x := \overline{1} & \{x = 1\} \\
\textsf{if } (x > 0) \textsf{ then} & \{1 = 1\} \\
\quad z := 1 & \{z = 1\} \\
\textsf{else} & \{0 = 1\} \\
\quad z := 0 & \{z = 1\} \\
\textsf{fi} & \{z = 1\}
\end{array}
$$

Also the preconditions of the three assignments are clear, since they're both just the weakest precondition of the assign statement and the postconditions.

$$
\begin{array}{ll}
 & \{T\} \\
x := \overline{1} & \{x = 1\} \\
\textsf{if } (x > 0) \textsf{ then} & \\
\quad z := 1 & \{z = 1\} \\
\textsf{else} & \\
\quad z := 0 & \{z = 1\} \\
\textsf{fi} & \{z = 1\}
\end{array}
$$

Actually, now that I think about it, the postconditions of the two branches are unnecessary also; they need to be $z = 1$ because that's the postcondition of the whole if statement.

$$
\begin{array}{ll}
 & \{T\} \\
x := \overline{1} & \{x = 1\} \\
\textsf{if } (x > 0) \textsf{ then} & \\
\quad z := 1 & \\
\textsf{else} & \\
\quad z := 0 & \\
\textsf{fi} & \{z = 1\}
\end{array}
$$

We're left with just the postcondition of $x = 1$ on the first assign statement. But $x = 1 \Leftrightarrow \exists x_0.[x_0/x]T \wedge x = [x_0/x]1 = sp(x := \overline{1}, T)$, so we could figure that one out also.

$$
\begin{array}{ll}
 & \{T\} \\
x := \overline{1} & \\
\textsf{if } (x > 0) \textsf{ then} & \\
\quad z := 1 & \\
\textsf{else} & \\
\quad z := 0 & \\
\textsf{fi} & \{z = 1\}
\end{array}
$$

We need the overall pre- and post-conditions for the whole program, because that's what we're trying to prove, so that's the best we can do. Turns out, being able to remove "some" of the conditions was an understatement. We got rid of all of them...

So do we never have to write out proof outlines again? Well, in the future, we will leave out some of the conditions that can be figured out using wp and sp (sorry, if you haven't finished HW3 yet, you still have to write out the whole proof every time). So what's the catch? Oh, right. We still haven't talked about the wp and sp of loops...

1.2 Problems Calculating the wp or sp of a Loop

How would we calculate $wp(W, q)$ where $W = \textsf{while } e \textsf{ do } s \textsf{ od}$?

- Let $w_0 = \neg e \wedge q$ and for all $k \geq 0$, define $w_{k+1} = e \wedge wp(s, w_k)$.
- If the loop runs for exactly n iterations, then $wp(W, q)$ is w_n.
- But in general, W might run for any number of iterations, so $wp(W, q) = w_0 \vee w_1 \vee \ldots$
- If this infinitely-long disjunction collapses somehow, then we can write $wp(W, q)$ finitely.
- e.g., if $w_{k+1} \Rightarrow w_k$ for when $k \geq 5$, then $wp(W, q) = w_0 \vee w_1 \vee w_2 \vee w_3 \vee w_4 \vee w_5$.
- Or, if all the w_k are parameterized by k (i.e., $w_k = P(k)$ for some P), then $wp(W, q) = \exists n.P(n)$.
- But in general, we don't know either of these facts.
- (sp has the same problem)
- So we'll give up on finding the wp and sp of loops in general (which also means we won't totally be able to erase annotations in proof outlines).

2 Proving Hoare Triples with Loops

We'll still need a rule for proving Hoare triples with loops. Here it is:

$$
\frac{\vdash \{p \wedge e\}\ s\ \{p\}}{\vdash \{p\}\ \textsf{while } e \textsf{ do } s \textsf{ od}\ \{p \wedge \neg e\}}\ (\textsc{While})
$$

- p needs to be true before and after s (remember: we might need to run s again!), so it's an *invariant*. We'll call it the *loop invariant*.
- Unlike with other statements, it's not always obvious what the right loop invariant is.
- We can always use something like T, but this probably isn't helpful (then we just get $\neg e$ as a postcondition of the loop, and we may need a stronger postcondition than that to prove stuff later).
- In general, a loop invariant needs to:
  - Be true at the start of the loop (maybe obtained by weakening some condition). Possibly we can also add some loop initialization code to make the invariant true.
  - Be provable after s knowing else nothing but e.
  - When anded with $\neg e$, imply the postcondition we want coming out of the loop.

2.1 Semantics of Invariants

Remember we expanded $\mathsf{while}\ e\ \mathsf{do}\ s\ \mathsf{od}$ into $\mathsf{if}\ e\ \mathsf{then}\ s; \mathsf{while}\ e\ \mathsf{do}\ s\ \mathsf{od}\ \mathsf{else\ skip\ fi}$. Does this play well with our definition of the rule for While?

$$\dfrac{\dfrac{\overline{\vdash \{p \wedge e\}\ s\ \{p\}} \qquad \dfrac{\cdots}{\vdash \{p\}\ \mathsf{while}\ e\ \mathsf{do}\ s\ \mathsf{od}\ \{p \wedge \neg e\}}}{\vdash \{p \wedge e\}\ s; \mathsf{while}\ e\ \mathsf{do}\ e\ \mathsf{od}\ \{p \wedge \neg e\}}\ (\textsc{Seq}) \qquad \dfrac{}{\vdash \{p \wedge \neg e\}\ \mathsf{skip}\ \{p \wedge \neg e\}}\ (\textsc{Skip})}{\vdash \{p\}\ \mathsf{if}\ e\ \mathsf{then}\ s; \mathsf{while}\ e\ \mathsf{do}\ s\ \mathsf{od}\ \mathsf{else\ skip\ fi}\ \{p \wedge \neg e\}}\ (\textsc{If})$$

In fact, it pretty much constrains what the rule for while has to be.

2.2 Example 1

$$\begin{array}{rl} t \triangleq & s := \overline{0}; \\ & k := \overline{0}; \\ & \mathsf{while}(k < n)\ \mathsf{do} \\ & \quad s := s + k; \\ & \quad k := k + \overline{1} \\ & \mathsf{od} \end{array}$$

We want to show $\vdash \{n \geq 0\}\ t\ \{s = sum(0, n)\}$, where $sum(0, n) = 0 + \cdots + n - 1$. We need a loop invariant.

Attempt 1: $p = (s = sum(0, k))$. So we need to show that

1	$\vdash \{s = sum(0,k) \wedge k < n\}\ s := s + k\ \{s - k = sum(0,k) \wedge k < n\}$	Assignment
2	$\vdash \{s = sum(0,k) \wedge k < n\}\ s := s + k\ \{s = sum(0,k) + k \wedge k < n\}$	Consequence 1
3	$\vdash \{(s = sum(0, k+1))\}\ k := k + \overline{1}\ \{s = sum(0,k)\}$	Assignment
4	$\vdash \{(s = sum(0,k) + k) \wedge k < n\}\ k := k + \overline{1}\ \{s = sum(0,k)\}$	Consequence 3
5	$\vdash \{p \wedge k < n\}\ s := s + k; k := k + \overline{1}\ \{p\}$	Sequence 2, 4
...	$\vdash \{n \geq 0\}\ s := \overline{0}; k := \overline{0}\ \{p\}$	Assignment, Sequence, Weakening
...	$\vdash \{n \geq 0\}\ t\ \{p \wedge k \geq n\}$	While, Sequence
...	$\vdash \{n \geq 0\}\ t\ \{s = sum(0, n)\}$	????

The problem is we can't make the last step because $s = sum(0,k) \wedge k \geq n \not\Rightarrow s = sum(0, n)$. We need $k = n$.

The loop invariant is *too weak*: it doesn't let us show the postcondition.

Let's try including more info about k.

Attempt 2: $p = (s = sum(0, k)) \wedge k = 0$. So we need to show that

1	$\vdash \{s = sum(0,k) \wedge k = 0 \wedge k < n\}\ s := s + k\ \{s - k = sum(0,k) \wedge k = 0 \wedge k < n\}$	Assignment
2	$\vdash \{(s = sum(0,k) + k = sum(0,k+1)) \wedge k = 0 \wedge k < n\}\ k := k + \overline{1}\ \{s = sum(0,k) \wedge k = 0\}$	??????

This invariant is *too strong*: we can't show that it's actually an invariant (holds after the loop body) (because it isn't).

Clearly we don't need to know in the loop body that $k = 0$. What we want to know after is that $k = n$. We know from the fact we've exited the loop that $k \geq n$. So if we also know $k \leq n$, then we know $k = n$.

Attempt 3: $p = (s = sum(0,k) \wedge k \leq n)$. So we need to show that

1	$\vdash \{p \wedge k < n\}\ s := s + k\ \{s - k = sum(0,k) \wedge k \leq n \wedge k < n\}$	Assignment
2	$\vdash \{(s = sum(0,k) + k = sum(0,k+1)) \wedge k + 1 \leq n\}\ k := k + \overline{1}\ \{s = sum(0,k) \wedge k \leq n\}$	Assignment
3	$\vdash \{(s = sum(0,k+1)) \wedge k \leq n \wedge k < n\}\ k := k + \overline{1}\ \{s = sum(0,k) \wedge k \leq n\}$	Consequence 2
4	$\vdash \{p \wedge k < n\}\ s := s + k; k := k + \overline{1}\ \{p\}$	Sequence 1, 3
...	$\vdash \{n \geq 0\}\ s := \overline{0}; k := \overline{0}\ \{p\}$	Asst, Seq, Cons.
...	$\vdash \{n \geq 0\}\ t\ \{p \wedge k \geq n\}$	While, Sequence
...	$\vdash \{n \geq 0\}\ t\ \{s = sum(0,n)\}$	Consequence

At the end, we have

$$s = sum(0,k) \wedge k \leq n \wedge k \geq n \Rightarrow s = sum(0,k) \wedge k = n \Rightarrow s = sum(0,n)$$

2.3 Proof Outlines

When writing proof outlines, we'll include loop invariants explicitly. Note that we can use strongest post-conditions (i.e. the forward assignment rule) to avoid having preconditions like $(0 = 0)$.

	$\{n \geq 0\}$
$s := \overline{0};$	$\{n \geq 0 \wedge s = 0\}$
$k := \overline{0};$	$\{n \geq 0 \wedge s = 0 \wedge k = 0\} \Rightarrow \{s = sum(0,k) \wedge k \leq n\}$
$\{\textbf{inv}\ s = sum(0,k) \wedge k \leq n\}$	
while$(k < n)$ do	$\{s = sum(0,k) \wedge k \leq n \wedge k < n\} \Rightarrow \{s + k = sum(0,k+1) \wedge k + 1 \leq n\}$
$\quad s := s + k;$	$\{s = sum(0,k+1) \wedge k + 1 \leq n\}$
$\quad k := k + \overline{1}$	$\{s = sum(0,k) \wedge k \leq n\}$
od	$\{s = sum(0,k) \wedge k \leq n \wedge k \geq n\} \Rightarrow \{s = sum(0,n)\}$

2.4 Loop Invariants and Programming

Sometimes thinking about loop invariants can also help you write correct programs in the first place. To see this, let's say we only have the outline below:

$$\begin{array}{ll} t \triangleq & \ldots \\ & \text{while}(k < n)\ \text{do} \\ & \quad \ldots \\ & \text{od} \end{array}$$

and we know we want the specification of the program to be $\vDash \{n \geq 0\}\ t\ \{s = sum(0,n)\}$. We might figure that we want the loop invariant to be $s = sum(0,k) \wedge k \leq n$. We need to set things up so that it holds at the beginning. First, n can be any nonnegative value, including 0, so the only thing we can initialize k to so that the invariant holds going into the loop is 0. That means we need to initialize s to $sum(0,0) = 0$.

$$\begin{array}{l} s := \overline{0}; \\ k := \overline{0}; \\ \text{while}(k < n)\ \text{do} \\ \quad \ldots \\ \text{od} \end{array}$$

Now we need to write the code of the loop so it maintains the invariant. There are a few ways we could do this, but as long as at the end of the loop body, we still have $s = sum(0, k) \wedge k \leq n$, we know the program is correct. We could also have done

$$
\begin{array}{l}
s := \overline{0}; \\
k := \overline{0}; \\
\mathsf{while}(k < n)\ \mathsf{do} \\
\quad k := k + \overline{1}; \\
\quad s := s + k - \overline{1} \\
\mathsf{od}
\end{array}
$$

If we wanted the postcondition to be $s = sumi(0, n)$, where $sumi(0, n) = 0 + \cdots + n$, this also tells us how to edit the program and/or invariants. We could use the fact that $sumi(0, n) = sum(0, n + 1)$ and try to get the condition at the end of the loop to be $s = sum(0, k) \wedge k = n + 1$. But if p is the loop invariant, we still just have $p \wedge k \geq n$ at the end of the loop. Maybe we should change the loop condition to $k < n + 1$, so we get $p \wedge k \geq n + 1$, and then the loop invariant $s = sum(0, k) \wedge k \leq n + 1$ would work.

$$
\begin{array}{rl}
t \triangleq & s := \overline{0}; \\
& k := \overline{0}; \\
& \mathsf{while}(k < n + 1)\ \mathsf{do} \\
& \quad s := s + k; \\
& \quad k := k + \overline{1} \\
& \mathsf{od}
\end{array}
$$

3 Strategies for Finding Loop Invariants

Unlike with finding the weakest (liberal) preconditions and strongest postconditions of loop-free programs, there's unfortunately no algorithm for finding invariants of loops. Often, especially at first, you may need to do several iterations of guessing an invariant and trying it out (like we did above) before finding the right one. As you do more of these, you'll start to see patterns that will help you guess the right invariant more easily. There are also some tricks and heuristics that can help, and these are the topic of the rest of this set of notes.

3.1 Replacing a Constant by a Variable

We did this one above: we had a constant n in the postcondition $s = sum(0, n)$ and we replaced the constant n with the variable k. In this case, k was a very useful choice because it happened to be equal to the loop index. What if we just chose an arbitrary variable, say i, that doesn't even appear in the program? We want i to be between 0 and n, so let's include that in the condition as well. Let the loop invariant be $s = sum(0, i) \wedge 0 \leq i \leq n$.

The loop invariant needs to be maintained by the body, which increments s by k. So after the increment, $s = sum(0, i) + k$. We need to change i such that $s = sum(0, i)$ again. The best way to do this is if we have, before, that $i = k$, and we then increment i. We can actually change the program to include i if we want.

$$
\begin{array}{l}
s := \overline{0}; \\
k := \overline{0}; \\
i := k; \\
\mathsf{while}(k < n)\ \mathsf{do} \\
\quad s := s + k; \\
\quad k := k + \overline{1}; \\
\quad i := k \\
\mathsf{od}
\end{array}
$$

But we quickly notice we're just keeping i in sync with k, so we may as well just use k instead of i in the loop invariant.

Notice that there are two constants in our postcondition: n and 0. We could also choose to replace 0 with a variable $0 \leq j \leq n$. So our loop invariant is $s = sum(j, n) \wedge 0 \leq j \leq n$. At the end, we need $j = 0$, so we want j to start at n and decrement to 0. Then what we need to establish on the loop body would be something like:

$$\{s = sum(j, n) \wedge 0 \leq j \leq n \wedge k < n\}\ s := s + k; k := k + \overline{1}; j := j - \overline{1}\ \{s = sum(j, n) \wedge 0 \leq j \leq n\}$$

The problem is that we've added k to s, but in order to keep $s = sum(j, n)$, we needed to add $j - 1$ to s. But we can't have $k = j - 1$ because k is increasing while j is increasing. So we would quickly realize this choice of loop invariant isn't going to work, at least assuming we can only make small changes to the program.

On the other hand, if we're coming up with the loop invariant first and then writing the program based on it (like in Section 2.4), we could come up with a perfectly correct program with this loop invariant:

$s := \overline{0};$
$j := n;$
while$(j > 0)$ **do**
  $j := j - 1;$
  $s := s + j$
od

But if the program is already written for us and we're trying to prove it correct, we need to replace n with a variable.

Replacing a constant by a variable is a simple version of the more general notion of adding or modifying the parameters of a predicate. Other versions of this technique include other changes:

Replace occurrences of	A constant A constant-valued expression A variable An expression	with	A new variable An expression with one or more new variables

3.2 Weakening the Invariant: Deleting a Conjunct or Adding a Disjunct

Sometimes the full postcondition won't be true the whole time: in particular, if part of the postcondition is actually the loop's termination condition, we shouldn't include that in the loop invariant.

Example 2: Linear Search of an Array

$i := \overline{0};$
while $(i < size(a) \wedge a[i] \neq n)$ **do**
  $i := i + \overline{1}$
od

We want to show

$$\{T\}\ s\ \{(\forall j \in [0, i-1].a[j] \neq n) \wedge (i < |a| \rightarrow a[i] = j)\}$$

First of all, let's see if the postcondition just works as the loop invariant. This means proving

$$\{(\forall j \in [0, i-1].a[j] \neq n) \wedge (i < |a| \rightarrow a[i] = j) \wedge i < size(a) \wedge a[i] \neq n\}\ i := i + \overline{1}\ \{(\forall j \in [0, i-1].a[j] \neq n) \wedge (i < |a| \rightarrow a[i] = j)\}$$

The first conjunct isn't hard to see: we know that for all $j \in [0, i-1]$, $a[j] \neq n$ and we know that $a[i] \neq n$, so we know that for all $j \in [0, i]$, $a[i] \neq n$ and the condition holds when we increment i.

The second conjuct will give us more trouble (we can't even necessarily establish it at the beginning of the loop!). So let's get rid of it and make the loop invariant just $\forall j \in [0, i-1].a[j] \neq n$. Why can we do this without preventing us from proving the postcondition? Because the other conjunct is implied by the negation of the loop condition.

$$\begin{array}{ll} & (\forall j \in [0, i-1].a[j] \neq n) \wedge \neg(i < |a| \wedge a[i] \neq n) \\ \Rightarrow & (\forall j \in [0, i-1].a[j] \neq n) \wedge (i \geq |a| \vee a[i] = n) \\ \Rightarrow & (\forall j \in [0, i-1].a[j] \neq n) \wedge (i < |a| \rightarrow a[i] = n) \end{array}$$

The full proof outline would then be:

	$\{T\}$
$i := \overline{0}$;	$\{i = 0\}$
$\{\textbf{inv}\ \forall j \in [0, i-1].a[j] \neq n\}$	
while $(i < size(a) \wedge a[i] \neq n)$ do	$\{(\forall j \in [0, i-1].a[j] \neq n) \wedge (i < \lvert a\rvert \wedge a[i] \neq n)\} \Rightarrow \{(\forall j \in [0, i].a[j] \neq n)\}$
$\quad i := i + \overline{1}$	$\{(\forall j \in [0, i-1].a[j] \neq n)\}$
od	$\{(\forall j \in [0, i-1].a[j] \neq n) \wedge \neg(i < \lvert a\rvert \wedge a[i] \neq n)\}$
	$\Rightarrow \{(\forall j \in [0, i-1].a[j] \neq n) \wedge (i < \lvert a\rvert \rightarrow a[i] = n)\}$

3.2.1 Adding a Disjunct

We can also think of part of what we did for Example 1 as weakening the condition: $k = n$ is part of the postcondition we wanted. We weakened that to $k \leq n$ (i.e., $k < n \vee k = n$).

3.3 Strengthening the Invariant

Sometimes we actually need the invariant to be stronger in order to prove something about the loop body or re-establish the invariant. For example, if we want total correctness in the array search example, we also need to know that $0 \leq i$, even though this isn't necessary to establish the postcondition ($i < |a|$ is ensured by the loop condition). **Note, of course**, that this doesn't actually establish total correctness because we haven't proven that the loop terminates (we'll see how to do this in a couple weeks). It just shows that there won't be an error.

	$[\ T\ ]$
$i := \overline{0}$;	$[i = 0] \Rightarrow [0 \leq i]$
$[\ \textbf{inv}\ 0 \leq i \wedge \forall j \in [0, i-1].a[j] \neq n\ ]$	
while $(i < size(a) \wedge a[i] \neq n)$ do	$[0 \leq i \wedge (\forall j \in [0, i-1].a[j] \neq n) \wedge (i < \lvert a\rvert \wedge a[i] \neq n)]$
	$\Rightarrow [0 \leq i + 1 \wedge (\forall j \in [0, i].a[j] \neq n)]$
$\quad i := i + \overline{1}$	$[0 \leq i \wedge (\forall j \in [0, i-1].a[j] \neq n)]$
od	$[(\forall j \in [0, i-1].a[j] \neq n) \wedge (i < \lvert a\rvert \rightarrow a[i] = n)]$

Example 3: Fibonacci

We want to verify a program to compute Fibonacci numbers, defined as

$$\begin{aligned} \mathit{fib}(0) &= 0 \\ \mathit{fib}(1) &= 1 \\ \mathit{fib}(n+2) &= \mathit{fib}(n) + \mathit{fib}(n+1) \end{aligned}$$

Here's the program, which we'll call s:

```
i := 0̄;
a := 0̄;
b := 1̄;
while (i < n) do
   c := a + b;
   a := b;
   b := c;
   i := i + 1̄
od
```

We want $\{n \geq 0\}\ s\ \{a = \mathit{fib}(n)\}$.
Let's try just replacing n with i in the postcondition:

	$\{n \geq 0\}$
$i := \overline{0};$	$\{i = 0\}$
$a := \overline{0};$	$\{i = 0 \wedge a = 0 = \mathit{fib}(i)\}$
$b := \overline{1};$	$\{i = 0 \wedge a = 0 = \mathit{fib}(i)\}$
$\{\textbf{inv}\ a = \mathit{fib}(i) \wedge i \leq n\}$	
while $(i < n)$ do	
$\quad c := a + b;$	
$\quad a := b;$	
$\quad b := c;$	
$\quad i := i + \overline{1}$	
od	

The loop invariant holds initially, but then we set a to b and we know nothing about b, so we have no hope of re-establishing the invariant! What do we need to know about b? Well, since it's assigned to a and i is incremented, we should hope it's equal to $\mathit{fib}(i+1)$.

	$\{n \geq 0\}$
$i := \overline{0};$	$\{i = 0\}$
$a := \overline{0};$	$\{i = 0 \wedge a = 0 = \mathit{fib}(i)\}$
$b := \overline{1};$	$\{i = 0 \wedge a = 0 = \mathit{fib}(i) \wedge b = 1 = \mathit{fib}(i+1)\}$
$\{\textbf{inv}\ a = \mathit{fib}(i) \wedge b = \mathit{fib}(i+1) \wedge i \leq n\}$	
while $(i < n)$ do	$\{a = \mathit{fib}(i) \wedge b = \mathit{fib}(i+1) \wedge i < n\}$
	$\Rightarrow \{b = \mathit{fib}(i+1) \wedge a + b = \mathit{fib}(i+2) \wedge i < n\}$
$\quad c := a + b;$	$\{b = \mathit{fib}(i+1) \wedge c = \mathit{fib}(i+2) \wedge i < n\}$
$\quad a := b;$	$\{a = \mathit{fib}(i+1) \wedge c = \mathit{fib}(i+2) \wedge i < n\}$
$\quad b := c;$	$\{a = \mathit{fib}(i+1) \wedge b = \mathit{fib}(i+2) \wedge i < n\}$
$\quad i := i + \overline{1}$	$\{a = \mathit{fib}(i) \wedge b = \mathit{fib}(i+1) \wedge i \leq n\}$
od	$\{a = \mathit{fib}(i) \wedge b = \mathit{fib}(i+1) \wedge i \leq n \wedge i \geq n\} \Rightarrow \{a = \mathit{fib}(n)\}$

4 Example 4: Binary Search

This program searches an array a for a value n using binary search.

```
l := 0̄;
r := size(a);
m := (l + r)/2̄;
while (l < r ∧ a[m] < n) do
   if (n < a[m]) then
      r := m;
   else
      l := m + 1;
   fi;
   m := (l + r)/2;
od
```

Our precondition is $\mathit{Sorted}(a) \wedge |a| > 0$. Our postcondition is

$$a[m] = n \vee (l + 1 = |a| \wedge a[l] < n) \vee a[l] < n < a[l+1] \vee (l = 0 \wedge n < a[l])$$

It may take a few minutes to convince yourself that this is the right postcondition.

We'll make a few weakenings: let's replace $l + 1$ with r. We'll also replace a couple of $<$ signs with $\leq$. How do you know which ones? This takes some trial and error, unfortunately.

$$p \triangleq l \leq m < r \wedge ((r = |a| \wedge a[l] \leq n) \vee (a[l] \leq n < a[r]) \vee (l = 0 \wedge n < a[l]))$$

First of all, does this, together with termination of the loop, imply the postcondition?

$$\begin{array}{ll} & p \wedge (l \geq r-1 \vee a[m] = n) \\ \Rightarrow & (a[m] = n \wedge p) \vee (p \wedge l \geq r-1) \\ \Rightarrow & a[m] = n \vee (l = m = r-1 \wedge (r = |a| \wedge a[l] \leq n) \vee a[l] \leq n < a[r] \vee (l = 0 \wedge n < a[l])) \\ \Rightarrow & a[m] = n \vee (l = m \wedge (l+1 = |a| \wedge a[l] \leq n) \vee a[l] \leq n < a[l+1] \vee (l = 0 \wedge n < a[l])) \\ \Rightarrow & a[m] = n \vee (l+1 = |a| \wedge a[l] < n) \vee a[l] < n < a[l+1] \vee (l = 0 \wedge n < a[l]) \end{array}$$

Why were we able to turn the two instances of $a[l] \leq n$ back into $a[l] < n$? Because by $l = m$, if $a[l] = n$, then $a[m] = n$, which is already covered by the first disjunct.

OK, so let's see if we can establish and preserve the invariant.

	$\{Sorted(a)\}$
$l := \overline{0}$;	
$r := size(a)$;	
$m := (l+r)/\overline{2}$;	$\{l = 0 \wedge r = \|a\| \wedge m = (l+r)/2\}$
$\{\textbf{inv}\ p\}$	
while $(l < r-1 \wedge a[m] \neq n)$ **do**	$\{p \wedge l < r-1 \wedge a[m] \neq n\}$
if $(n < a[m])$ **then**	$\{p \wedge l < r-1 \wedge a[m] \neq n \wedge n < a[m]\}$
$r := m$;	
else	$\{p \wedge l < r-1 \wedge a[m] \neq n \wedge n \geq a[m]\}$
$l := m+1$;	
fi;	$\{[((l+r)/2)/m]p\}$
$m := (l+r)/\overline{2}$;	
od	

To complete the proof outline, we just need to show that both branches of the **if** establish $[(l+r)/2/m]p$, which means that the invariant will be preserved after the assignment:

$$\{p \wedge l < r-1 \wedge a[m] \neq n \wedge n < a[m]\}\ r := m\ \{[((l+r)/2)/m]p\}$$

and

$$\{p \wedge l < r-1 \wedge a[m] \neq n \wedge n \geq a[m]\}\ l := m+1\ \{[((l+r)/2)/m]p\}$$

where

$$[(l+r)/2/m]p = l \leq (l+r)/2 < r \wedge ((r = |a| \wedge a[l] \leq n) \vee (a[l] \leq n < a[r]) \vee (l = 0 \wedge n < a[l]))$$

The first conjunct definitely holds $l \leq (l+r)/2 < r$, and the second conjunct is the same as in p, so effectively we just have to show that the second part of p is preserved by the two branches.

Let's consider each disjunct of p separately:

- $r = |a| \wedge a[l] \leq n$. Then we can consider which branch we're in:
  - $n < a[m]$. Then, after the assignment, $r = m$ and so $n < a[r]$. We still have $a[l] \leq n$, so p holds.
  - $n > a[m]$. Then, after the assignment, $l = m+1$, and so $n > a[l-1]$, which means $n \geq a[l]$. We still have $r = |a|$, so p still holds.
- $a[l] \leq n < a[r]$. Again, consider which branch we're in:
  - $n < a[m]$. Then, after the assignment, $r = m$ and so $n < a[r]$. We still have $a[l] \leq n$, so p holds.
  - $n > a[m]$. Then, after the assignment, $l = m+1$, and so $n > a[l-1]$, which means $n \geq a[l]$. We still have $n < a[r]$, so p still holds.
- $l = 0 \wedge n < a[l]$. Then $n < a[m]$, so we're in the first branch and don't change l, so the invariant is preserved.