Slide 1

Slide 2

CS 2112

Recitation 8
Loop Invariants

October 22 / 23, 2024

Slide 3

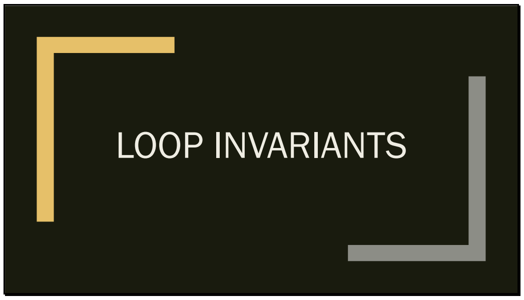
Agenda

- Loop Invariants
- Bucket Sort
- Binary Search
- Exponentiation

Reminders

- TA Evaluations
- A4 is a thing

Slide 4



A brief review of loop invariants, which you saw in lecture

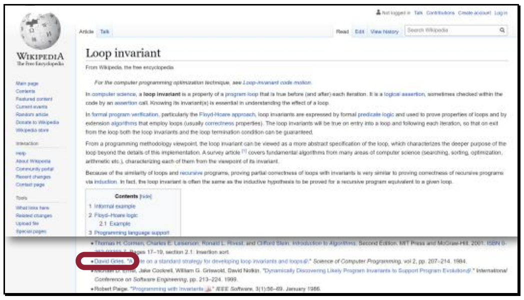
Slide 5

The first question we inevitably get when talking about loop invariants is, “why do we study loop invariants”

Slide 6

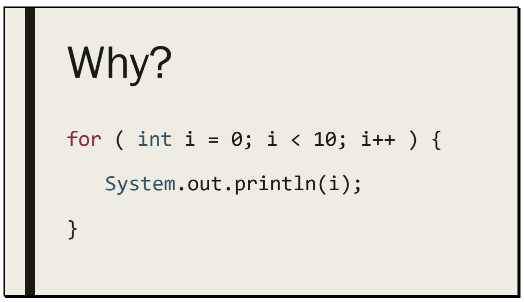
Prof. David Gries, former CS 2110 professor, had a particularly famous response to this question

Slide 7



In fairness, when the Wikipedia page for a topic cites you as the source, perhaps you've earned the right to a bit of sass

Slide 8



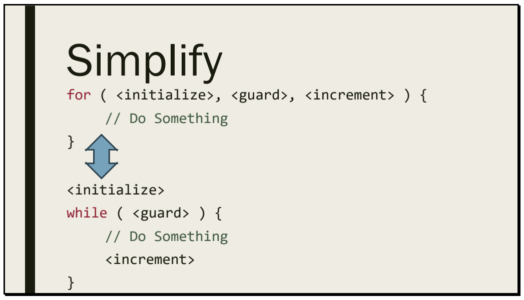
But this is a fair question. When the loops we write are simple, it's obvious what they do just by looking at them.

Slide 9

However, as loops get more complicated, this is no longer true. (Pictured here is a random loop I found in the Linux source code. I have no idea what it does. It just looks long and scary)

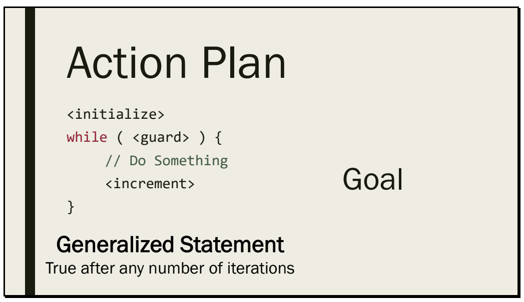
The point of a loop invariant is to provide a formal framework for us to convince ourselves (and others) that the loop we wrote actually does what we claim it does.

Slide 10



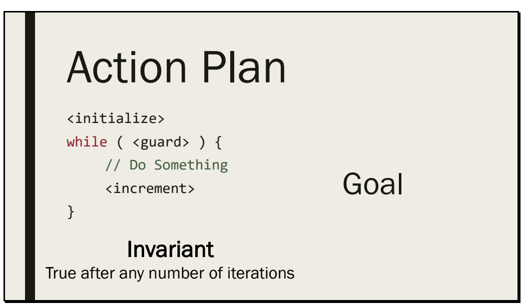
Since all for loops can be written as a while loop, for simplicity, we will only be looking at while loops today.

Slide 11



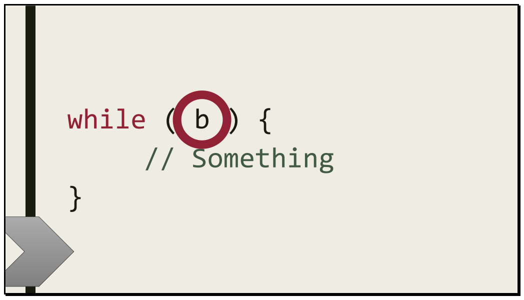
The biggest reason loops are so hard to reason about is that they can run any number of times. So, in the proud tradition of students everywhere, when a problem is too hard, we'll give up and tackle an easier problem.
The key idea is to come up with some generalized statement that is true after any number of iterations of the loop. By doing so, we can then reason without needing to worry about how many times the loop runs, sidestepping the hard part.

Slide 12



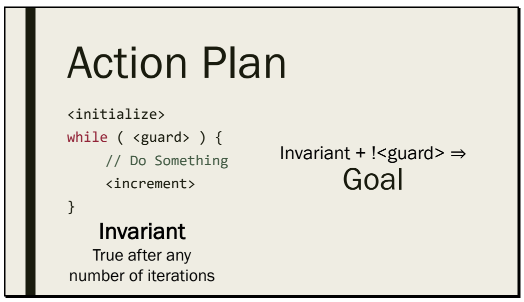
You have probably guessed by now that the general statement is what we call the "Loop Invariant"

Slide 13



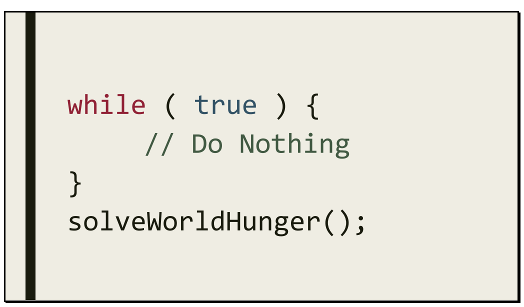
When we exit a loop, the thing we know for a fact is that the loop guard ("b" in this case) must be false, or else the loop would still be running.

Slide 14



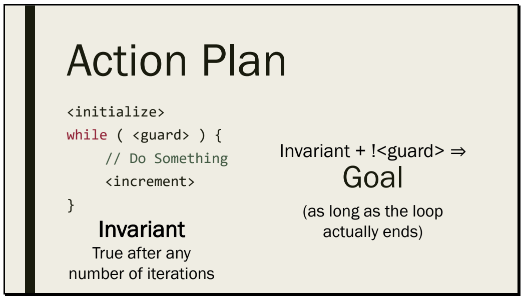
Therefore, our goal is to show that the loop invariant together with the negation of the loop guard proves the goal we want our loop to fulfill.

Slide 15



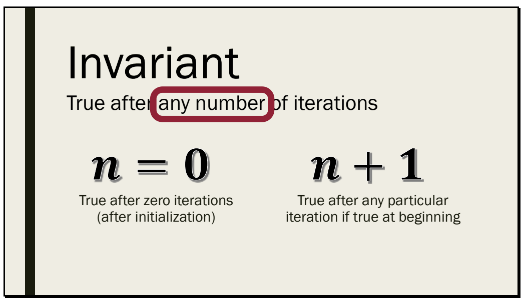
The other caveat is that a loop which doesn't end will never exit, and thus our loop guard will never be false.

Slide 16



Therefore, a complete proof also needs to show that the loop actually ends.

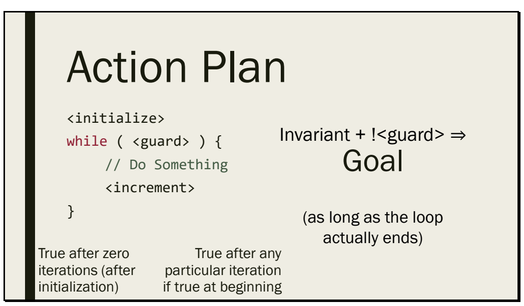
Slide 17



To show that the invariant is true after any number of iterations, we first must show it is true for the smallest number of iterations a loop can make, which is zero (NOT one!). Then, we can show that it's true for any n+1 iterations if it was true for n iterations. This is called induction.

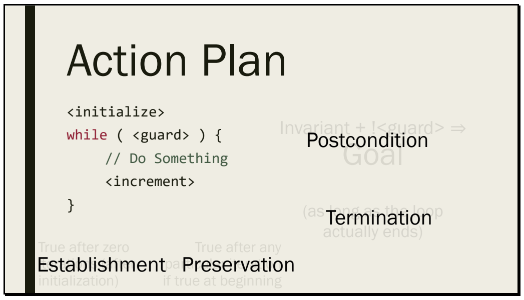
(in fact, a loop invariant proof is exactly a proof by induction, where the loop invariant is our inductive hypothesis)

Slide 18



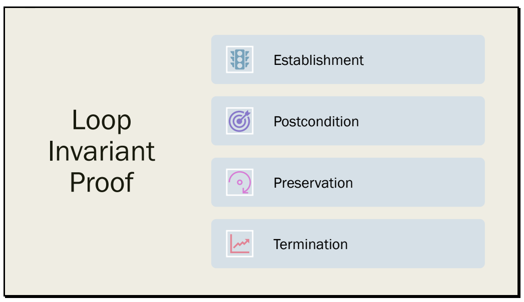
Thus here are the four steps we need to prove to complete our proof of the loop's correctness

Slide 19



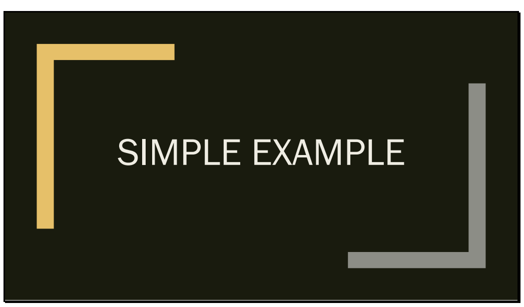
We can assign fancy names to these steps

Slide 20

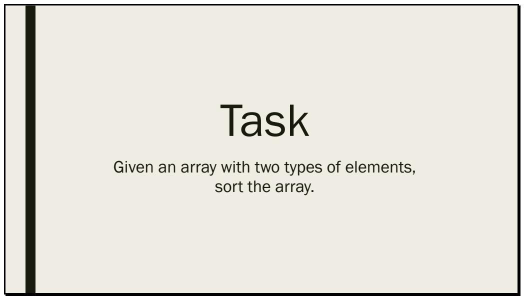


And hopefully you recognize these names as the four parts you were taught in lecture.

Slide 21

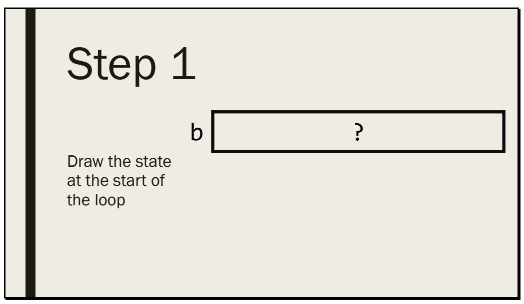


Slide 22



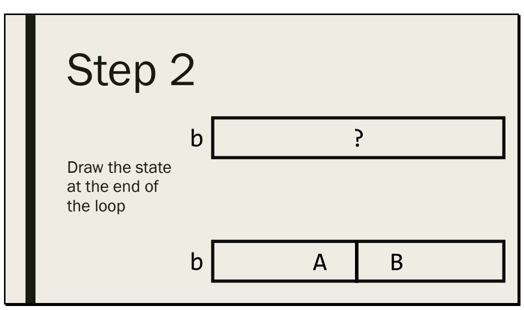
Here's an actual interview question I got from Microsoft when applying for my internship there.
I'm going to show you a foolproof four-step plan to write any loop correctly.

Slide 23



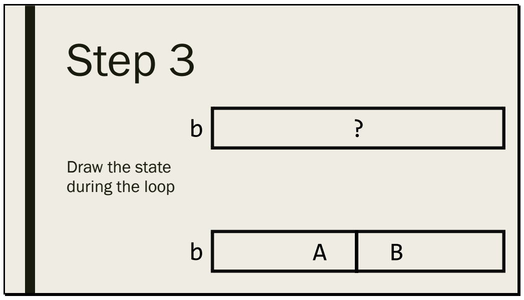
I'm using pictures here instead of words for intuition. CS 2110 used to teach loop invariants exclusively with pictures, and while we do expect 2112 students to be able to formalize their invariants in writing, I do still find the pictures useful for organizing my thoughts.
In this picture, at the start of the loop, nothing is sorted yet, so everything is marked with a "?", indicating that the invariant makes no promises about its contents (other than the obvious stuff, like that there's two types of things in here)

Slide 24



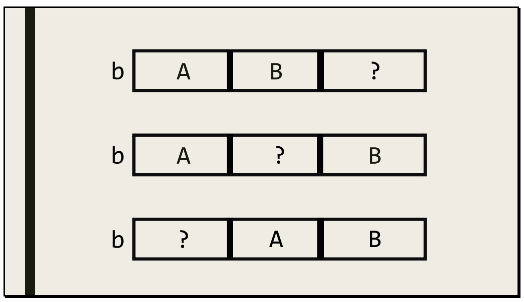
At the end of the loop, the two types of things should be properly sorted (marked by "A" and "B").

Slide 25



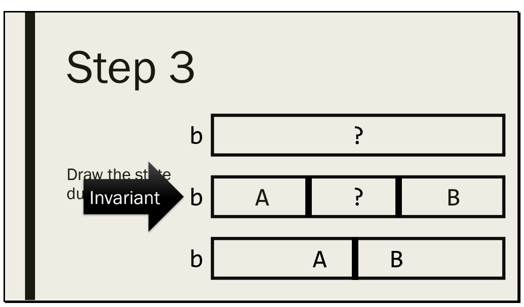
Now we need to visualize how the loop should transition from the start to the end.

Slide 26



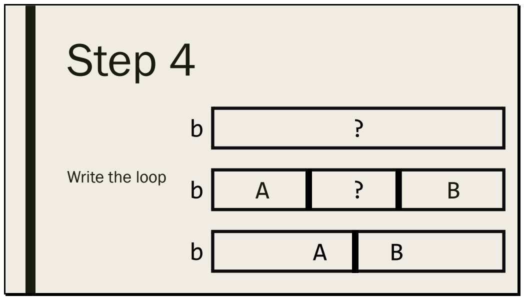
There's usually multiple ways to make that transition. In these three examples, you can see the sorted section growing in from either end of the array. The point is, pick one.

Slide 27



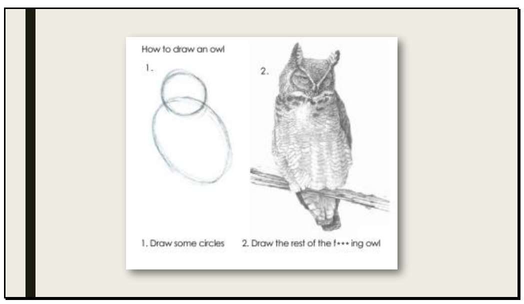
Unsurprisingly, this intermediate state you picked is going to form your loop invariant.

Slide 28



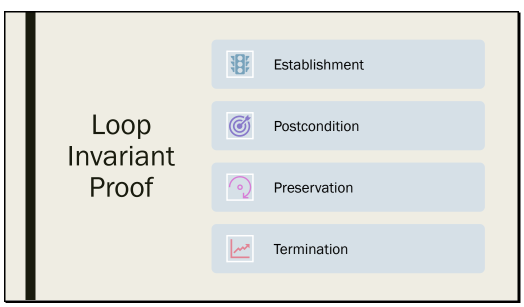
Final step is to write the loop

Slide 29



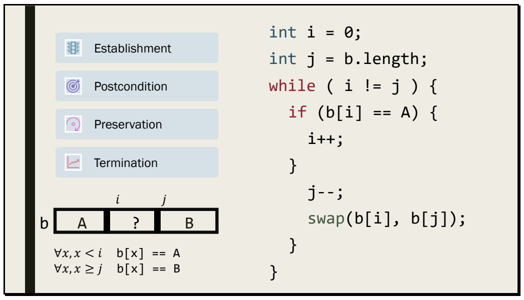
Admittedly, this final step does not sound useful. But I promise that we're onto something here…

Slide 30



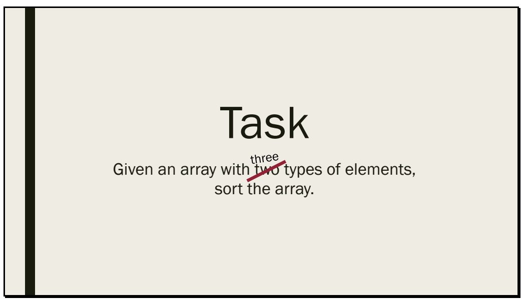
Because the four steps of the loop invariant proof can now guide our loop writing process.

Slide 31



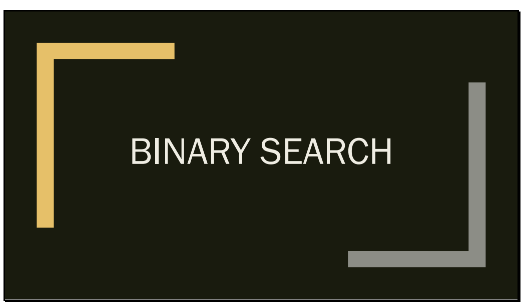
Establishment forces `i` and `j` to be certain values at the start. The post condition lets us reason about what must be true at the end of the loop (`i == j`), so we negate that to form the loop guard. Then, termination tells us how to make progress (`i++` or `j--`) and then preservation forces the rest of the loop to be correct.
This was the exact loop I wrote for my interviewer. I got the job.

Slide 32

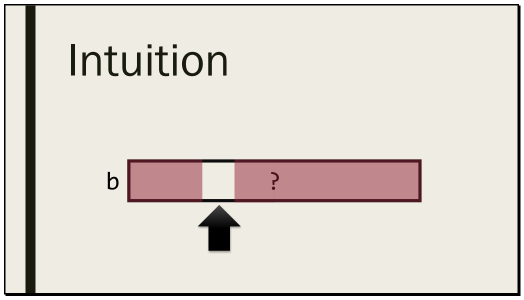


Now you try. This was the followup question my interviewer asked me. Same setup, but there's three types of elements now.

Slide 33

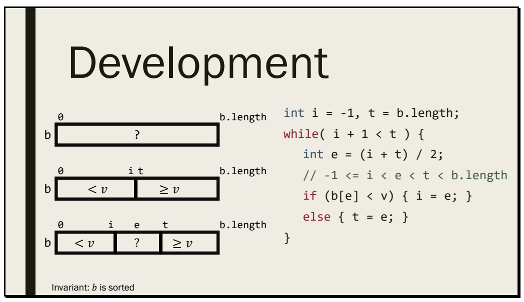


Slide 34



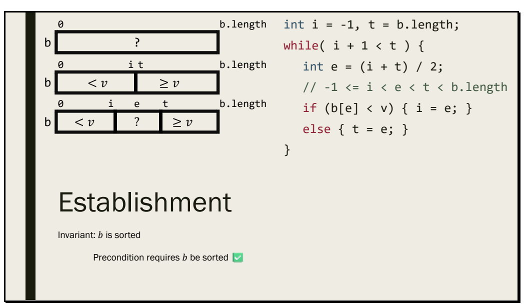
You saw this in a previous discussion; binary search finds things in a sorted array quickly by jumping to the middle of the remaining half at each step.

Slide 35



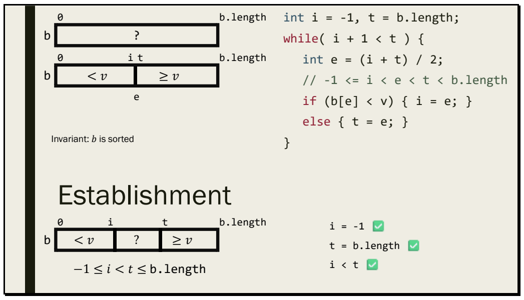
We can use the same loop invariant formalism to write the loop.
Next, we'll practice proving the loop's correctness.

Slide 36



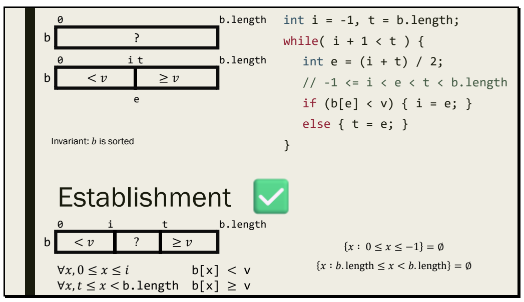
The invariant requires b be sorted, but that's also the precondition of the binary search, so trivially this is satisfied.

Slide 37



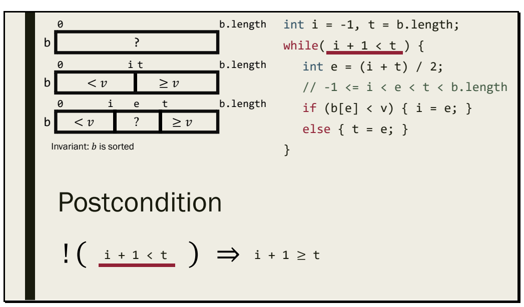
Next, we have bounds on i and t, but these are also trivially true at the start of the loop.

Slide 38



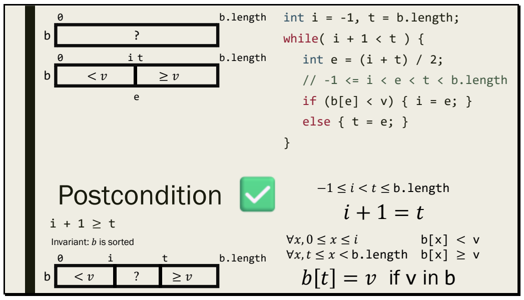
Most importantly, our invariant guarantees certain chunks of the array are smaller or larger than v, the target value. Luckily, at the start, we've chosen values such that these chunks of the array are empty, meaning it's trivially true.
Establishment tends to be the easy part to prove, in general.

Slide 39



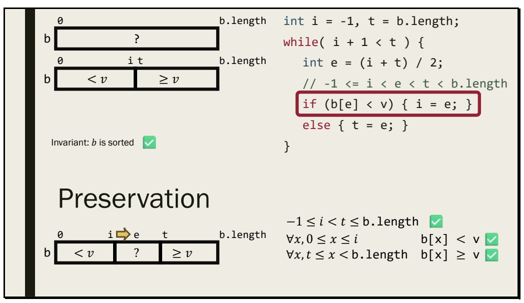
Next, the postcondition tells us the loop guard must be false, which means we negate the part inside the loop guard.

Slide 40



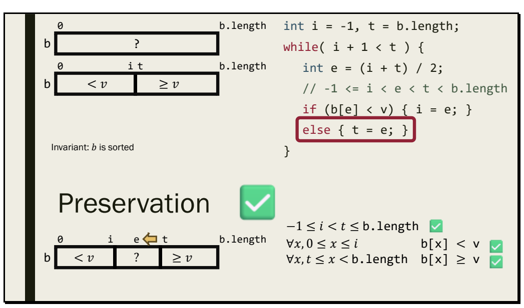
Using this fact, we can see that given our bounds requires $i < t$, yet $i + 1 \geq t$, we must conclude that $i + 1 = t$. In other words, i is the index of an element immediately before t.
But then, our invariant tells us b is sorted, and the element at index i is $< v$, while the element at index t is $\geq v$.
If i comes immediately before t, then the element at t must be v, since if v can't be earlier in the array since i must be less than v, and it can't be later since the array is sorted.
This means we've found the position of v.

Slide 41



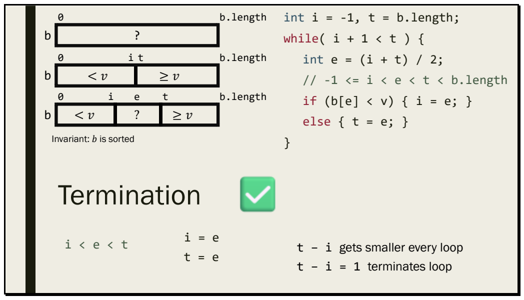
There's two cases inside the loop we need to examine for preservation. If b[e] $< v$, we see that we lift i up to e. Since e < t, the bounds are still correct. Since b[e] $< v$ and b is sorted, all the things up to and including position e must be $< v$, which means we can safely lift i up to it. Finally, we don't modify t at all (nor do we modify the array), so the invariant about t and b being sorted are all maintained.

Slide 42



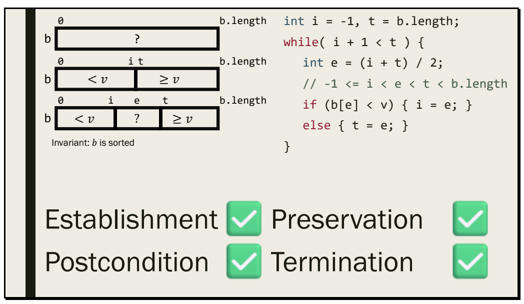
By the same logic, moving t down to e if b[e] $\geq v$ is also safe (convince yourself this is true).
Thus, we have proven preservation.

Slide 43



Finally for termination, note that the quantity $t - i$ must strictly decrease every loop since we either set i up to e or t down to e, and $i < e < t$. Therefore, $t - i$, an integer, decreases strictly every iteration, and when it equals 1, the loop ends. Since it starts at a value ≥ 1 and will never go below 1 (by the invariant), termination is guaranteed.

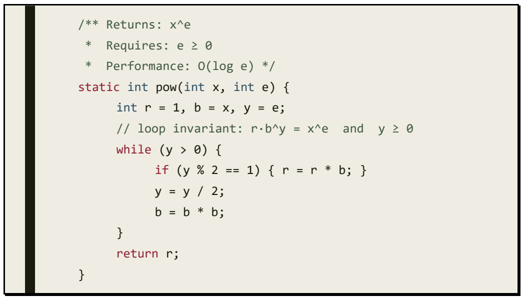
Slide 44



This completes the loop invariant proof.

Slide 45

Slide 46



Now you try. Here's a loop that exponentiates. The invariant is given to you. Prove its correctness.

Slide 47