CSE 331
Software Design & Implementation

Hal Perkins

Winter 2023

Lecture 3 – Reasoning About Loops

Administrivia

- Reminder: HW1 out now, due Tuesday night, 11 pm
  - 1 late day (max) possible, but you *really* don't want to use it

- Reminder: be sure to read lecture notes as well as slides for this and yesterday's classes

- Reminder: readings on the calendar – *Pragmatic Programmer* (PP) and *Effective Java* (EJ)
  - The readings give "item" or "topic" numbers in the books because they are organized that way; not pages or chapters
  - Free access to these books online via UW library's institutional license – see the syllabus or other course resources for access details
  - You'll get the most out of class if you make a first pass over the readings before the associated class

Loops

- Reference: new lecture notes that cover this material but with more details and explanations. See the course calendar for link.

- Apology: these slides generally don’t have fancy fonts or colors
  - If it helps, imagine seeing all of this done on a much wider whiteboard… ☺

Reasoning about loops

So far, two things made our code reasoning fairly straightforward:

1. When running the code, each statement executed 0 or 1 times

2. (Therefore,) trivially the code always terminates

Neither of these hold once we have loops (or recursion)

- Will consider the key ideas with while-loops
- Introduces the essential and much more general concept of an *invariant*
- Will mostly ignore prove-it-terminates; brief discussion later

What’s a loop?

To keep things simple we will only consider while loops

```
while (B)
  S;
```

We don’t lose anything by this (and it simplifies our life). A for-loop for (init, test, step) S is simply a convenient way of writing

```
init;
while (test) {
  S;
  step;
}
```

Other loops like for (x: collection) S can be rewritten similarly.

Loops and Proofs

We want to analyze loops much the same way we handle other code by inserting assertions between statements to keep track of the program state

```
{ P }
while (B) {
  { assertion }    ☜ What do we know here?
  S;
}
{ Q }
```

So let’s do something similar to what we did for IF statements…

Loops and Proofs

What do we know the first time around the loop?

```
{ P }
while (B) {
  { P ∧ B }
  S;
  { Q1 }
}
{ Q }
```

If the loop stops now, all we need to do is prove { Q1 ∧ !B } => { Q } to show we have what we want.

But what if the loop isn't done after one iteration? What do we know the next time around?

Loops and Proofs

What do we know the *second* time around the loop?

```
{ P }
while (B) {
  ~~{ P ∧ B }~~ { Q1 ∧ B }
  S;
  { Q2 }
}
{ Q }
```

If the loop stops now, all we need to do is prove { Q2 ∧ !B } => { Q } to show we have what we want.

But what if the loop isn't done after two iterations? What do we know the next time around?

Loops and Proofs

What do we know the *third* time around the loop?

```
{ P }
while (B) {
  ~~{ P ∧ B }~~ ~~{ Q1 ∧ B }~~ { Q2 ∧ B }
  S;
  { Q3 }
}
{ Q }
```

If the loop stops now, all we need to do is prove { Q3 ∧ !B } => { Q } to show we have what we want.

But what if the loop isn't done after three iterations? What do we know the next time around?

And what if the loop body is never executed? (B false initially)

Loop Invariants

The complication is that we need figure out something that is right regardless of whether the loop body executes 0, 1, 2, … times

```
{ P }
while (B) {
  { assertion }    ☜ What do we always know here?
  S;
}
{ Q }
```

The essence of dealing with loops is figuring out "what's true at the top of the loop body every time" – we call that the *(loop) invariant*.

Loop Initialization

Almost always a loop involves some initialization before the loop test, so our overall loop and its proof will look like this. We need to show the invariant holds every time we reach the start of the loop body

```
{ P }
initialization
{ inv }
while (B) {
  { inv ∧ B }
  S;
  { inv }
}
{ inv ∧ !B } => { Q }
```

☜ loop invariant, often abbreviated as { I } or { inv } once it’s been specified the first time

Loop Proofs

To prove that our loop works properly, we need to show:

1. That {inv} holds after the initialization finishes, right before loop condition B is evaluated the first time (i.e., {P} init {inv}), and
2. If {inv ∧ B} is true at the beginning of the loop body then {inv} is true after the loop body executes, and
3. {inv ∧ !B} => { Q }

Note: we sometimes need to add a bit of code after the loop to fix things up to establish { Q }, but we won't need to do this in our initial examples.

```
{ P }
initialization
{ inv }
while (B) {
  { inv ∧ B }
  S;
  { inv }
}
{ inv ∧ !B } => { Q }
```

Picking an invariant

Idea: capture the idea of “what is true each time around the loop when we have done part of the total job, but are not finished yet”

- This is the inventive/creative step in writing a loop; there is no automatic procedure for doing it
  - Requires “thinking” or “guessing”
  - You have been doing this all along – just never used these words
- Invariant needs to be strong enough so that we can prove what we need, but not so strong that the proof won’t work
- There may be many invariants that “work” but some might be easier to reason about than others

The BIG IDEA

Programming is a creative activity. With loops the creative part is coming up with the loop body and invariant. Doing this in tandem often makes it easier to come up with correct code (fewer bugs ☺).

Strategy (not the only way to do it, but usually very productive – try it!) Write a loop "inside-out" in this order:

1. Choose a loop invariant and write the loop body together
   - This is the inventive step
   - Very often a good loop invariant is a weaker version of the postcondition (in our stronger-weaker sense!)
2. Choose B (the loop condition) so that { inv ∧ !B } => { Q }
   - (maybe after adding a few statements after the loop if needed to make { Q } true – not used in our initial examples
3. Add initialization steps to get from { P } to { inv }

(notation: we'll use { pre: … } for {P} and { post:…} for {Q} in examples)

Example: sum

The first problem is trivial, but will help us understand the strategy

Problem: write a loop to set *sum* = 1 + 2 + … + *n*

What do we want?

{ post: sum = 1 + 2 + … + n }

{ pre: __________ } (we’ll figure this out later…}

Step 1: loop invariant and body

Problem: write a loop to set *sum* = 1 + 2 + … + *n*

{ post: sum = 1 + 2 + … + n }

Invent an invariant! Idea: Weaken the post condition to get
{ inv: sum = 1 + 2 + … + k-1 } for some k

- i.e., introduce a variable k and add it to sum each time through the loop body
- Loop body will be roughly (but check as we write/prove it)

```
sum = sum + k;
k = k + 1;
```

(why k-1 as the bound in inv instead of k? well, try both and tinker and see which one works best!)

Loop body

Remember that { inv ∧ B } holds before the loop and { inv } needs to be true at the end. Let's work it out, adding assertions between statements using forward reasoning:

```
{ inv ∧ B: sum = 1 + 2 + … + k-1 ∧ B }
sum = sum + k;
{ sum = 1 + 2 + …  + k }
k = k + 1;
{ sum = 1 + 2 + … + k-1 }    //  { inv !!! }
```

Step 2: loop condition

Program so far:

```
{ pre: _________ }
___________;                  // initalization (to be done later)
{ inv: sum = 1 + 2 + … + k-1 }
while (B) {                                  // B (to be done next)
  { inv ∧ B: sum = 1 + 2 + … + k-1 ∧ B }
  sum = sum + k;
  { sum = 1 + 2 + …  + k }
  k = k + 1;
  { inv: sum = 1 + 2 + … + k-1 }
}
{inv ∧ !B } => { post: sum = 1 + 2 + … + n }
```

Next: pick B such that { inv ∧ !B } => { post }

- This is mechanical: we need !B to be (k-1 = n) !!!
- So B is !(k-1 == n) (which we'll simplify to k-1!=n)

Step 3: Initialization

Program so far:

```
{ pre: _________ }
___________;                  // initalization (to be done next)
{ inv: sum = 1 + 2 + … + k-1 }
while (k-1 != n)
  { inv ∧ B: sum = 1 + 2 + … + k-1 ∧ k-1 != n }
  sum = sum + k;
  { sum = 1 + 2 + …  + k }
  k = k + 1;
  { sum = 1 + 2 + … + k-1 }
}
{inv ∧ !B } => { post: sum = 1 + 2 + … + n }
```

Initialization: also mechanical – set k and sum so inv holds

- sum = 0 and k = 1 will do nicely! (i.e., sum = 1 + 2 + … + k-1)
- When k = 1, this is an empty sequence 1 + … + 0 which is 0

Finishing up

Done. But is this always correct? i.e., is { pre } = { true } ?

- Not quite – the proof only works if n >= 0, so that's our precondition

```
{ pre:  n >= 0  }
sum = 0;
k = 1;
{ inv: sum = 1 + 2 + … + k-1 }
while (k-1 != n)
  { inv ∧ B: sum = 1 + 2 + … + k-1 ∧ k-1 != n }
  sum = sum + k;
  { sum = 1 + 2 + … + k-1 + k }
  k = k + 1;
  { sum = 1 + 2 + … + k-1 }
}
{inv ∧ !B } => { post: sum = 1 + 2 + … + n }
```

Recap

We spent a lot of effort to develop a simple loop. But we saw how the proof and loop construction techniques work together:

1. Discover (invent) invariant and develop loop body and proof
2. Calculate loop condition B from { inv ∧ !B } => { post }
3. Figure out initialization and precondition to establish { inv }

What we didn't consider: termination. Our proof only really shows "the code is correct provided that it terminates". Our examples will be simple enough that termination is not an issue, but a complete proof would need it (we'll ignore).

- (One way to do it: figure out some sort of number or expression that captures "how much more to do" and prove that it eventually reaches 0.)

More to come…

But first!

Administrivia

- HW1 due tomorrow night **11 pm** (not 11:59, not 11 pm Honolulu time, …)
  - Yes you can use one (1) late day *max* if you want to
  - NO you don't want to (if at all possible)….

- Discussion board:
  - Please try to use descriptive subject headings so others can easily see if there are already postings about things they have questions about. (Can we be more precise than HW7 Q42?)
    - So far pretty good this quarter – keep it up!
  - It's helpful if there is enough context in a posting to figure out what it's about without having to pull up a copy of the assignment/slide/etc. and read it.
    - Text is really great and compact compared to screen grabs if you can do that. ☺

Upcoming assignments and sections

- HW2 – loops, proofs, and invariants – posted later today. Due a week from Tuesday.

- HW3 – first programming problem and infrastructure shakedown cruise – out later this week and will be the main topic of sections
  - New “Do this before section this week” writeup on the resources page soon. Basically: install java 17, git, intellij.
  - Then sections this week will explain and walk through infrastructure and hw3 basics, and you can try on your computer at the same time.
    - Bring a laptop with installed software to section this week if you can!
  - More handouts on the web coming soon – watch for ed announcements.
    - The posted instructions are intended to work. If problems, use the class discussion board – don’t try random things you find on stackoverflow or web searches

New problem: max element in array

Problem: we are given an array *items* with *size* elements. Write a loop to store the largest value in that array in variable *max*.

First figure out the postcondition: we want

{ post: max = largest in items[0..size-1] }

New notation, which will be very handy for assertions about arrays: If b is an array, then b[i..j] is the section of the array containing b[i], b[i+1], …, b[j]. If j=i-1 the section is empty (e.g., b[0..-1]). If i=j, the section contains one element (e.g., b[2..2] is b[2]).

Since we are using assertions and proofs as design tools, not input to an automated proof checker, we'll allow ourselves to use informal but precise statements like "largest in a[i..j]" or "max of a[i..j]"

Step 1: loop invariant and body

Strategy: go through the array one element at a time and compare next element to previous largest value. If the new element is larger, update largest-value-seen-so-far. Now, need a loop invariant…

The postcondition is

  { post: max = largest in items[0..size-1] }

Let’s weaken this to get a plausible invariant

  { inv: max = largest in items[0..k-1] }

Notice: we’ve just added variables k and max to our code! This is how we discover what variables need to be declared in our code – create and name them when we discover things we need to store!

(Why k-1? why not k? k+1? This is part of the creative step – try out different possibilities and see which one(s) seem to work best to lead to clean code and a simpler proof.)

Loop body with assertions

```
{ inv: max = largest in items[0..k-1] }
if (max < items[k]) {
  { max = largest in items[0..k-1] ∧ max < items[k] }
  max = items[k];
  { max = largest in items[0..k] }
} else {
  // nothing needs updating (but need a place to write assertions)
  { max = largest in items[0..k-1] ∧ max >= items[k] }
       => { max = largest in items[0..k] }
}
{ max = largest in items[0..k] }
k = k + 1;
{ inv: max = largest in items[0..k-1] }
```

Step 2: loop condition

Program so far:

```
{ pre: ________ }
__________;                        // initalization (to be done later)
{ inv: max = largest in items[0..k-1] }
while (B) {                                  // B (to be done next)
  { inv: max = largest in items[0..k-1] }
  if (max < items[k]) {
    { max = largest in items[0..k-1] ∧ max < items[k] }
    max = items[k];
    { max = largest in items[0..k] }
  } else {
    // nothing needs updating
    { max = largest in items[0..k] }
  }
  { max = largest in items[0..k] }
  k = k + 1;
 { inv: max = largest in items[0..k-1] }
}
{inv ∧ !B } => { post: max = largest in items[0..size-1] }
```

Next: pick B such that { inv ∧ !B } => { post }

- Mechanical again: we need !B to be (k == size) !!!
- So B is !(k == size) (which we'll simplify to k!=size)

Now just wait a minute!!!

What's with picking a loop condition that has != in it??

```
        while (k != size) { … }
```

This code seems strange (which is a way of saying *I think it has to be wrong!*) Wouldn't it be safer to use something like k < size?

A: No. We want to be able to prove that { inv ∧ !B } => { post }. With k != size for the loop condition we get

{inv ∧ !B } = { max = largest in items[0..k-1] ∧ !(k != size) }
=> { post: max = largest in items[0..size-1] }

With k<size we would have have to show that when k >= size we really have k = size. Otherwise the proof doesn't work.

- (CSE 311 veterans might enjoy a chance to show off their proof-by-induction skills, but why complicate things if we don't have to?)

Step 3: Initialization

Program so far:

```
{ pre: ________ }
__________;                        // initalization (to be done next)
{ inv: max = largest in items[0..k-1] }
while (k != size) {
  { inv: max = largest in items[0..k-1] }
  if (max < items[k]) {
    { max = largest in items[0..k-1] ∧ max < items[k] }
    max = items[k];
    { max = largest in items[0..k] }
  } else {
    // nothing needs updating
    { max = largest in items[0..k] }
  }
  { max = largest in items[0..k] }
  k = k + 1;
  { inv: max = largest in items[0..k-1] }
}
{inv ∧ !B } => { post: max = largest in items[0..size-1] }
```

Initialization is also mechanical. Pick values for max and k to establish inv

- We can set max = items[0] and k = 1

Finishing up

This code works for non-empty arrays, so we get our precondition:

```
{ pre: size > 0 }
k = 1;
max = items[0]
{ inv: max = largest in items[0..k-1] }
while (k != size) {
  { inv: max = largest in items[0..k-1] }
  if (max < items[k]) {
    { max = largest in items[0..k-1] ∧ max < items[k] }
    max = items[k];
    { max = largest in items[0..k] }
  } else {
    // nothing needs updating
    { max = largest in items[0..k] }
  }
  { max = largest in items[0..k] }
  k = k + 1;
  { inv: max = largest in items[0..k-1] }
}
{inv ∧ !B } => { post: max = largest in items[0..size-1] }
```

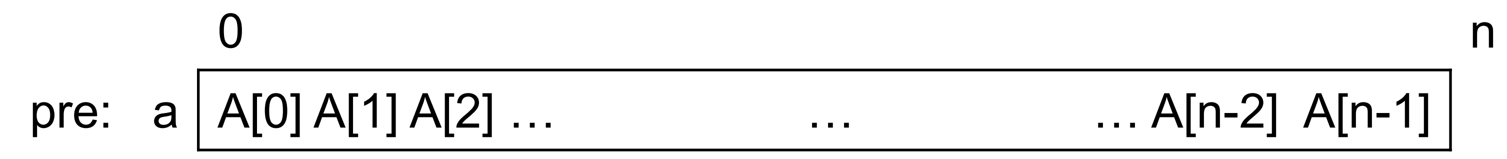
Loose ends

- The code we've developed works for non-empty arrays.
- What if size is 0 (i.e., the array is empty)?
  - That's a fairly philosophical question: what is the largest value in an empty array?
  - If the code needs to work for an empty array (i.e., size==0), the specification needs to say what to do in that case. A couple of plausible solutions:
    - Specify Integer.MIN_VALUE as the largest value in an empty array
    - Throw an appropriate exception if size is not positive
- What about size < 0? Probably an error (illegal argument exception?) A good specification should say what happens.

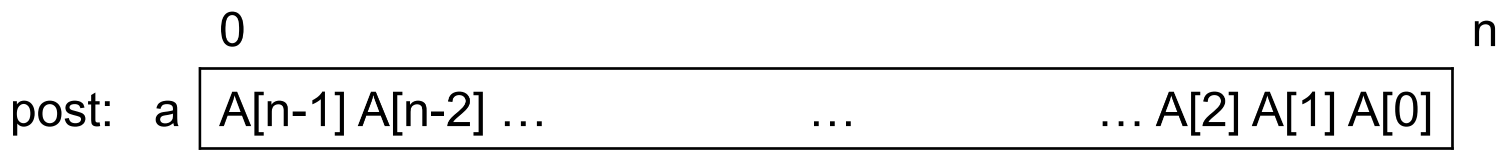
New problem: reverse an array

Problem: given an array a with n elements (a[0..n-1]), reverse the order of the elements in a.

This is another fairly simple problem, but we want to build a correct solution and avoid all off-by-one errors. We start with:



We want:

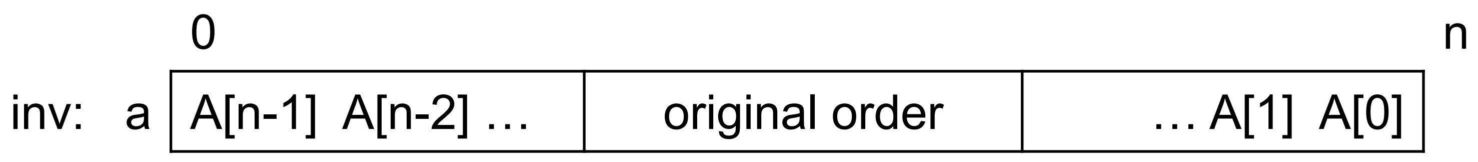


Notation: we'll use A[i] (capitalized) for the value in position i in the original array

Step 1: loop invariant and body

We’ll use the “obvious” strategy: start out by swapping the first and last elements (a[0] and a[n-1]), then swap the next pair of elements (a[1] and a[n-2]), and continue until we’re done.

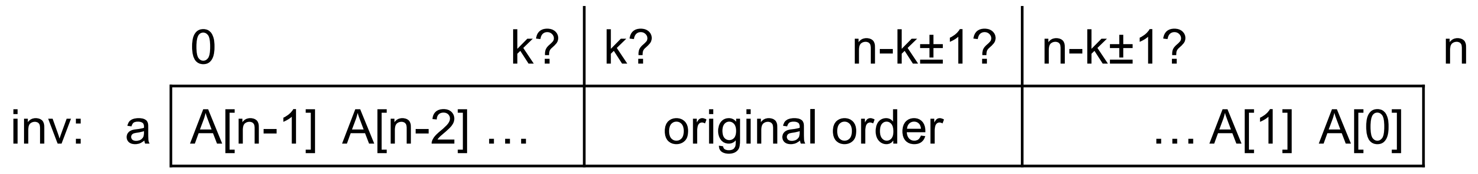
The invariant looks like this:



(Note: it’s often very useful to think/design with diagrams. We don’t have to restrict ourselves to the a[i..j] notation all the time, but we’ll usually want to use that later to be precise.)

Array section boundaries

We need to introduce variables to keep track of the boundary positions between the different sections of the array. A typical solution would be to introduce a variable k and do some arithmetic:

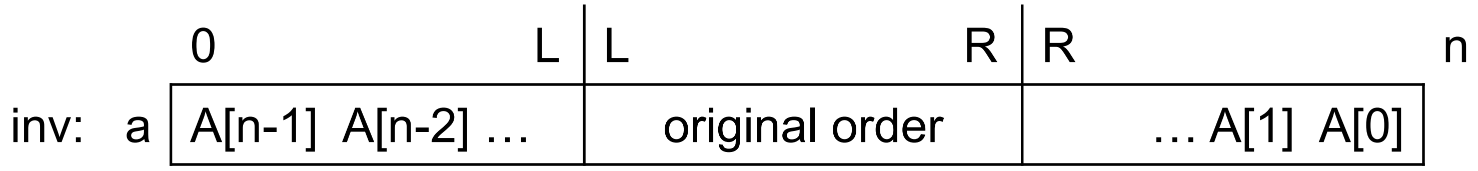


Yuck! (😱) The n-k±1 boundary is a glaring invitation for an off-by-one bug! Not to mention, when do we stop? k = n/2? k = n/2±1? k > n/2? ?? Can we do something simpler?

Yes!!

Simpler array section boundaries

Just name the boundaries – don’t have to calculate them, do we?

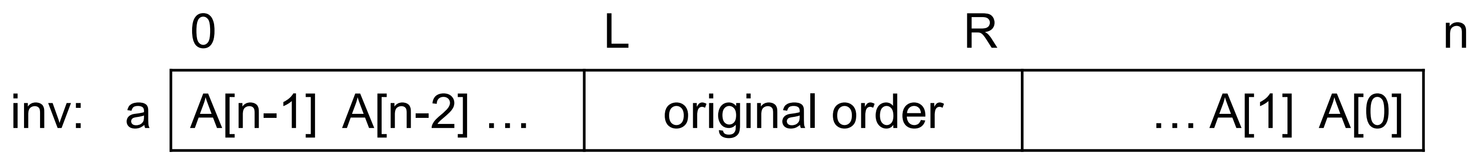


We still have to decide whether the L and R variables mark the ends of the reversed areas or the ends of the area that is still in the original order. What should we choose?

The answer is Yes!! We do need to choose something and stick with it, but it’s not always obvious which choice will be best. So try out things that seem plausible, sketch the code and the proof that results, and then pick the choice that makes things work best.

Invariant for reverse array

After trying alternatives, let’s label the unswapped area boundaries:

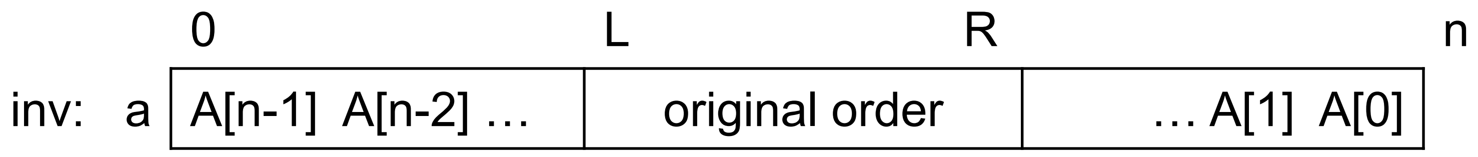


The body of the loop looks like this:

```
{ inv }
swap(a[L], a[R])          // use swap(…) notation for design
L = L + 1;                // no ++ or --.  we don’t have a
R = R - 1;                //    proof rule for ++ or --.
{ inv }
```

Step 2: loop condition

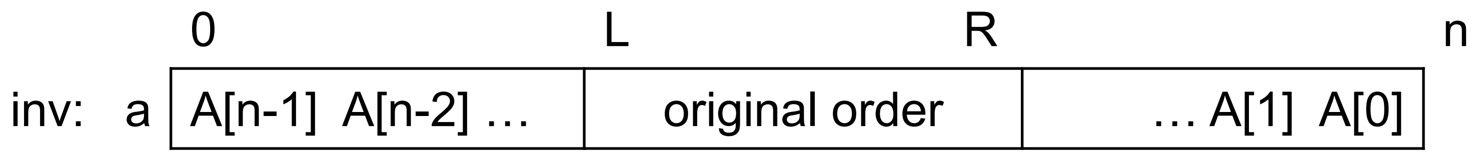
When are we done? When there are 0 or 1 elements in a[L..R] !



```
initialize
{ inv }
while ( L < R )  {
  { inv }
  swap(a[L], a[R])
  L = L + 1;
  R = R - 1;
  { inv }
}
```

Step 3: initialize

Initially, the entire array is the ”original order” middle section, so set L=0, R=n-1 to start



```
L = 0; R = n-1;
{ inv }
while ( L < R)  {
  { inv }
  swap(a[L], a[R])
  L = L + 1;
  R = R - 1;
  { inv }
}
```

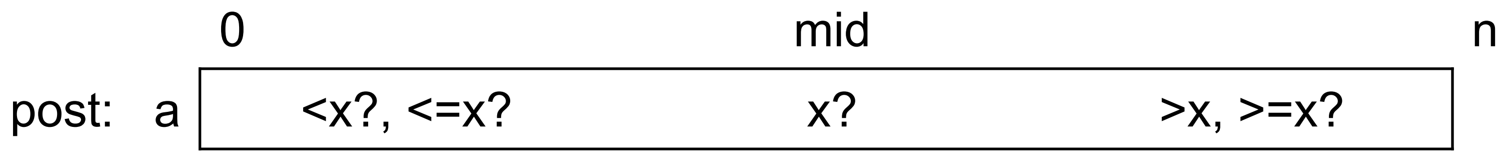
And we’re done! (proof that postcondition holds at the end is left to the reader. ☺)

New problem: binary search

Problem: We are given a value x and a sorted array a with n elements (i.e., a[0] <= a[1] <= … <= a[n-1]). Find the location of x in the array.

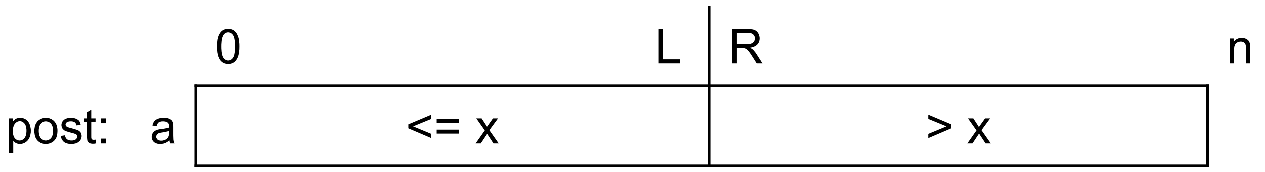
We'll use the usual strategy: look in the middle, compare to x, then eliminate half of the array from consideration based on the comparison. Quit when we've found x if it is present.

But if we do that in the "usual" way, we get a postcondition that is a mess: what if there are multiple copies of x in a, what if x is not present, what if x is larger or smaller than anything in a, …?



Binary search postcondition

We would like a postcondition that "works" regardless of whether x is in the array or not, whether or not there are multiple copies of x, and so forth. After trying several alternatives, let's pick this:



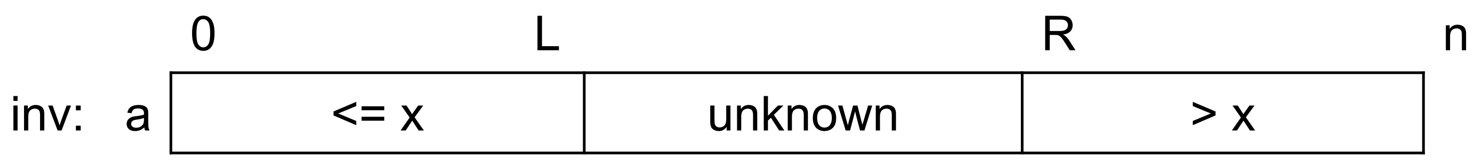
More precisely: { post: a[0..L] <= x ∧ a[R..n-1] > x ∧ L+1 = R }

If there are one or more copies of x in the array, they will be at a[L], a[L-1], … . The final regions a[0..L] or a[R..n-1] might be empty if x is smaller or larger than everything in the array. (We would have either L=-1 or R=n in those cases.)

Step 1: loop invariant and body

Weaken the postcondition to get the invariant:

a[0..L] <= x ∧ a[L+1..R-1] not searched ∧ a[R..n-1] > x



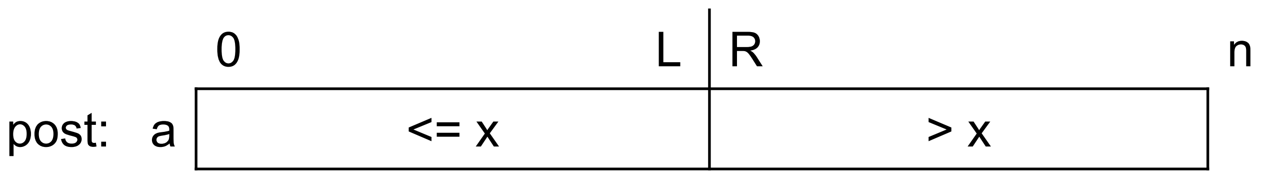
The loop is the usual one:

```
while ( ____________ ) {
  mid = (L + R) / 2;          // truncating division
  if (a[mid] <= x)
    L = mid;
  else  // a[mid] > x
    R = mid;
}
```

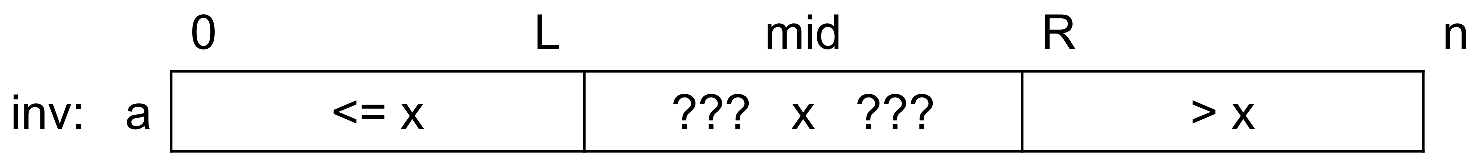
Whoa! Wait a minute!!!!

You can’t do that! You have to test a[mid] == x and STOP if it is!!!

Absolutely not!! remember we want a clean postcondition



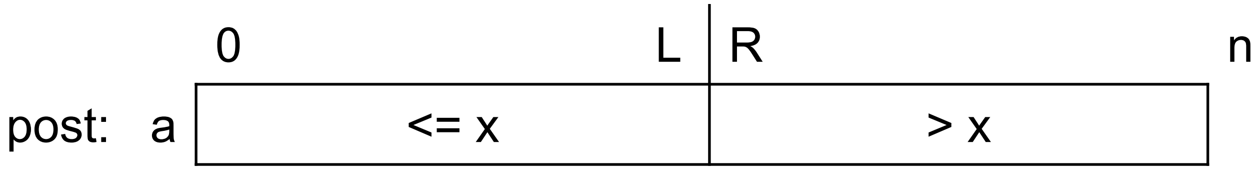
and if we stop in the middle, we get a mess like this!



(It also turns out that stopping in the middle does not reduce number of iterations enough to matter, but it makes *every* loop iteration slower because of the extra == test!)

Step 2: loop condition

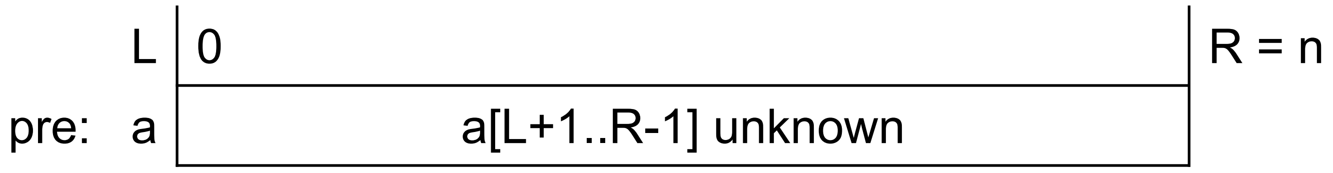
When are we done? When L and R are right next to each other. that gives us the loop condition.



```
while ( **L+1 != R** ) {
  mid = (L + R) / 2;
  if (a[mid] <= x)
    L = mid;
  else  // a[mid] > x
    R = mid;
}
```

Step 3: Initialization

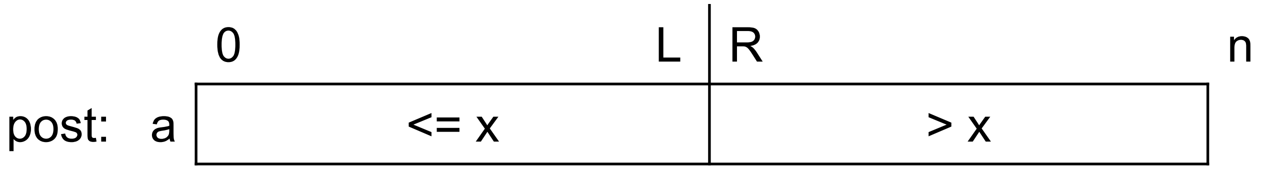
When we start out, a[0..n-1] is the “unknown” region, which is a[L+1..R-1] according to our invariant. Initialize L, R accordingly.



```
L = -1; R = n;            // initialize: a[0..L] and a[R..n-1] empty
while ( L+1 != R ) {
  mid = (L + R) / 2;
  if (a[mid] <= x)        // not a bug.  0 <= mid < n always
    L = mid;
  else  // a[mid] > x
    R = mid;
}
```

What do we know when we’re done?

When the loop terminates we have



- If x appears in the array, then a[L] = x, and if there are multiple copies, so is a[L-1], … .
- If x is larger than all array elements, a[R..n-1] is empty, R = n, and L = n-1.
- If x is smaller than all array elements, then a[0..L] is empty, L = -1, and R = 0.
- So x is in the array if

```
found = (L >= 0) && (a[L] == x);   // short-circuit && required
```

(Precondition footnote: works for non-empty arrays (n>0). Add code if needs to work for n==0.)

New problem: Dutch National Flag

Given an array of red, white, and blue pebbles, sort the array so the red pebbles are at the front, white are in the middle, and blue are at the end

- [Use only swapping contents rather than “count and assign”]

Edsgar Dijkstra

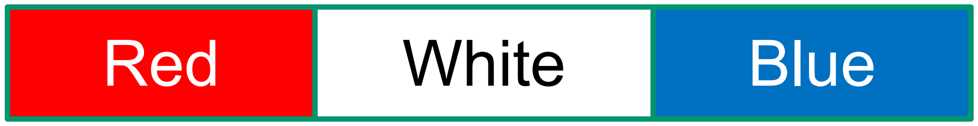
Pre- and post-conditions

Precondition: Any mix of red, white, and blue

Mixed colors: red, white, blue

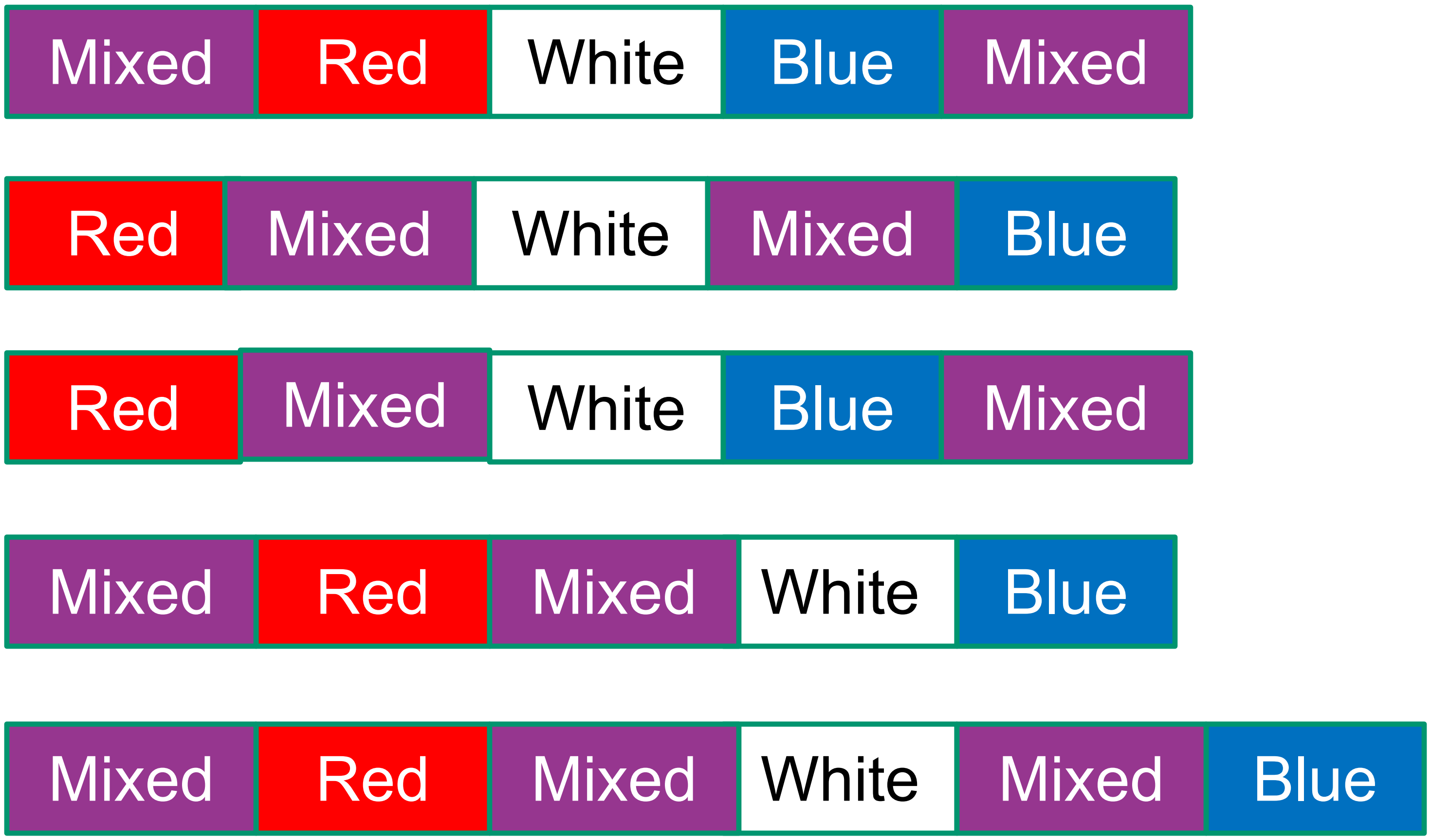
Postcondition:

- Red, then white, then blue
- Number of each color same as in original array



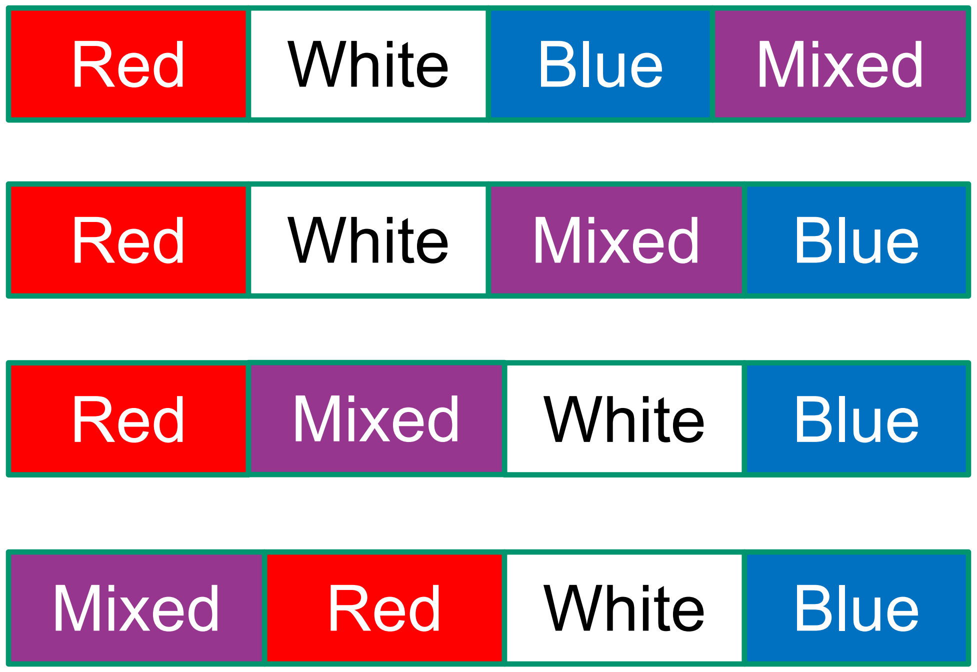
Step 1: loop invariant and body

Lots of possibilities. Here are some… ☺


Mixed Red White Blue Mixed
Red Mixed White Mixed Blue
Red Mixed White Blue Mixed
Mixed Red Mixed White Blue
Mixed Red Mixed White Mixed Blue

But wait!! There's more!!!

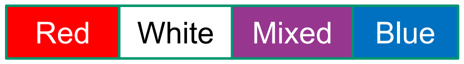
Simpler is very likely better. Let's minimize the number of regions



Middle two are probably more useful since we won't have to move red and blue pebbles once they are in place. We'll go with #2.

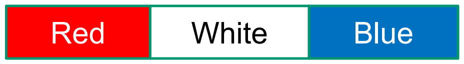
More precise, and then some code

- Precondition **`P`**: **`a`** contains **`r`** reds, **`w`** whites, and **`b`** blues
- Invariant: **`P ∧ 0 <= i <= j <= k <= n`**



```
∧ arr[0..i-1] is red
∧ arr[i..j-1] is white
∧ arr[j..k-1] is mixed
∧ arr[k..n-1] is blue
```

- Postcondition: **`P ∧ 0 <= i <= j = k <= n`**

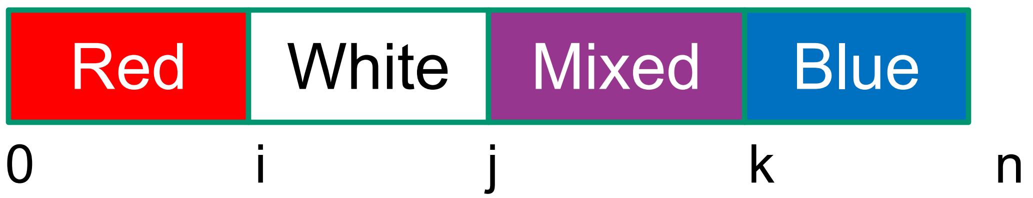


```
∧ a[0..i-1] is red
∧ arr[i..j-1] is white
∧ arr[k..n-1] is blue
```

- Initialization to establish the invariant: **`i=0; j=0; k=n;`**

Mixed colors: red, white, blue

The loop test and body (one pass)



```
i = 0; j = 0; k = n;
while(j!=k) {
  if(arr[j] == White) {
     j = j+1;
  } else if (arr[j] == Blue) {
     swap(arr[j],arr[k-1]);
     k = k-1;
  } else { // arr[j] == Red
     swap(arr[i],arr[j])
     i = i+1;
     j = j+1;
  }
}
```

Termination – what we skipped so far

- Two kinds of loops
  - Those we want to always terminate (normal case)
  - Those that may conceptually run forever (e.g., web-server)

- So, proving a loop correct usually also requires proving termination
  - We haven't been proving this: might just preserve invariant forever without test ever becoming false
  - Our Hoare triples say ***if*** loop terminates, postcondition holds
  - Our loops have been simple enough that termination has been obvious, so we've skipped it up to now (and will in hw also)

- How to prove termination (variants exist):
  - Map state to a natural number somehow (just "in the proof")
  - Prove the natural number goes down on every iteration
  - Prove test is false by the time natural number gets to 0

Termination examples

- Reverse array: size of the unprocessed part of the array (initially array length, decreases by 2 each time through the loop, done when unprocessed length is 0 or 1)
- Binary search: size of range still to be considered
- Dutch-national-flag: size of range not yet partitioned (**`k-j`**)
- Search in a linked list: length of list not yet considered
  - Don't know length of list, but goes down by one each time…
  - … unless list is cyclic in which case, termination not assured

Perspective – When do we need proofs?

- Most loops are so "obvious" that proofs are, in practice, overkill
  - `for(String name : friends) {...}`

- Don't write a loop if a library has what you need
  - You probably will rarely if ever need to write reverse – use a list container or library function that reverses arrays.
  - Use the library version of binary search, don't re-invent the wheel.

- Use a **`for`** loop when it makes sense (compact, easier to read, index variable declared locally, don't need a while loop to provide places to put assertions, etc.)
  - `for (init; test; step) {...}`

When to use proofs for loops

- Use logical reasoning when intermediate state (invariant) is unclear or edge cases are tricky or you need inspiration, etc.

- Use logical reasoning as an intellectual debugging tool
  - What *exactly* is the invariant?
  - Is it satisfied on every iteration?
  - Are you sure? Write code to check?
  - Did you check all the edge cases?
  - Are there preconditions you did not make explicit?

- You don't need this for easy loops. It can become *essential* for hard loops (or other tricky code).
  - Must include invariant as a comment in the code if it's tricky – otherwise how is someone reading the code supposed to understand *why* it works, *how* it works, and *why* it's correct?

Proofs, code, and tools

- Software tools that analyze programs using proof techniques are ubiquitous in industry these days
  - Many variations on the kinds of logics used and how the tools work and what they can discover

- Because of computability/decidability results (cf CSE 311), no tools can be both complete (always can answer right/wrong) and correct (always gives the right answer)
  - In practice static analysis tools try to find as many potential problems as possible without raising too many false alarms

- The ideas we've learned should help you take advantage of tools and give you better insight into what they are doing

Onward…

- Reasoning about programs and how to “talk about” what a program should do and what it means for software to be “correct” is fundamental.
- We will use these ideas repeatedly in specifications, design of data structures and abstractions, reasoning about correctness of implementations, testing, and many other places.
  - and when writing tricky loops. ☺

- But before we move on completely: hw2 will be out after class (problems and proofs with loops). Due next Tuesday, **11 pm**