Handout C2: Reasoning About Code (Hoare Logic)

CSE 331, Spring 2012
Written by Krysta Yousoufian
With material from Michael Ernst, Hal Perkins, and David Notkin

Contents

Introduction

In this handout you will learn how to formally reason about code. This process will enable you to prove to yourself and others that a given block of code works correctly.

Code reasoning fundamentals

Imagine it's your first internship, and you are asked to write a max() method for an IntList class. You write this code and bring it to your boss. She says, "Prove to me that it works." OK… how do you do that? You could (and surely would) run some tests on sample input, but there's effectively an infinite number of possible IntLists. Tests are useful, but they can't prove that your code works in all possible scenarios.

This is where reasoning about code comes in. Instead of running your code, you step back and read it. You ask: "What is guaranteed to be true at this point in the program, based on the statements before it?" Or, going in the in other direction: "If I want to guarantee that some fact Q is true at this point in the

program, what must be true earlier to provide that guarantee?" You've surely done some of this naturally. Now you'll learn to do it in a more structured way with techniques to help.

Assertions

Let's start with a simple code example:

```
x = 17;
y = 42;
z = x+y;
```

At each point before/after/in between statements, what do we know about the state of the program, specifically the values of variables? Since we're looking at this chunk of code in isolation, we don't know anything before it executes. After the first line executes, we know that `x = 17`. After the second line executes, we still know that `x = 17`, and we know that `y = 42` too. After the third line executes, we also know that `z = 17 + 42 = 59`. We annotate the code to show this information:

```
{ true }
x = 17;
{ x = 17 }
y = 42;
{ x = 17 Λ y = 42 }
z = x+y;
{ x = 17 Λ y = 42 Λ z = 59 }
```

> **An aside: notation**
>
> If this notation is unfamiliar to you:
>
> Λ means AND
>
> V means OR
>
> A way to remember which symbol is which is that the AND symbol looks like the letter A.

Each logical formula shows what must be true at that point in the program. Since we don't know anything at the beginning, only "true" itself must be true, so we simply write `{true}`.

Each of the lines with curly braces is an assertion. An **assertion** is a logical formula inserted at some point in a program. It is presumed to hold true at that point in the program. There are two special assertions: the precondition and the postcondition. A **precondition** is an assertion inserted prior to execution, and a **postcondition** is an assertion inserted after execution. In the example above, `{true}` is the precondition and `{ x = 17 Λ y = 42 Λ z = 59 }` is the postcondition. All other assertions are called intermediate assertions. They serve as steps as you reason between precondition and postcondition, kind of like the intermediate steps in a math problem that show how you get from problem to solution.

Forward and backward reasoning

The process we just followed is called **forward reasoning**. We simulated the execution of the program, considering each statement in the order they would actually be executed. The disadvantage of forward reasoning is that the assertions may accumulate a lot of irrelevant facts as you move through the program. You don't know which parts of the assertions will come in handy to prove something later and which parts won't. As a result, you often end up listing everything you know about the program.

This happens in forward reasoning because you don't know where you're trying to go – what you're trying to prove. But when we write a block of code, we usually have a clear idea of what's supposed to be true after it executes. In other words, we know the postcondition already, and we want to prove that the expected postcondition will indeed hold true given the appropriate precondition. For this reason, **backward reasoning** is often more useful than forward reasoning, though perhaps less intuitive.

In backward reasoning, you effectively push the postcondition up through the statements to determine the precondition. You start by writing down the postcondition you want at the end of the block. Then you look at the last statement in the block and ask, "For the postcondition to be true after this statement, what must be true before it?" You write that down, move up to the next statement, and ask again: "For the assertion after this statement to be true, what must be true before it?" You keep going until you've reached the top of the statement list. Whatever must be true before the first statement is the precondition. You have guaranteed that if this precondition is satisfied before the block of code is executed, then the postcondition will be satisfied afterward.

For example, let's look at this two-line block of code:

```
x = y;
x = x + 1;
{ x > 0 }
```

The postcondition is `x > 0`. We want to know what must be true beforehand for the postcondition to be satisfied. We start with the last statement: `x = x + 1`. If `x > 0` afterward, then the value assigned to x (namely, x+1) must be > 0 beforehand. We add this assertion:

```
x = y;
{ x + 1 > 0 }
x = x + 1;
{ x > 0 }
```

Now we look at the second-to-last statement: `x = y`. If `x + 1 > 0` after this statement, then `[the value assigned to x] + 1 > 0` beforehand. That is, `y + 1 > 0`. We add this assertion:

```
{ y + 1 > 0 }
x = y;
{ x + 1 > 0 }
x = x + 1;
{ x > 0 }
```

Since there are no more statements, we're done. We have proven that if `y + 1 > 0` before this block of code executes, then `x > 0` afterward.

Weakest precondition

In the example above, `y + 1 > 0` is not the only valid precondition. How about the precondition `y = 117`? Or `y > 100`? These, too, guarantee the postcondition. Technically they're correct, but intuitively they're not as useful. They are more restrictive about the values of y for which the program is

guaranteed to be correct. We usually want to use the precondition that guarantees correctness for the broadest set of inputs. Stated differently, we want the **weakest precondition:** the most general precondition needed to establish the postcondition. The terms "weak" and "strong" refer to how general or specific an assertion is. The weaker an assertion is, the more general it is; the stronger it is, the more specific it is. We write `P = wp(S,Q)` to indicate that P is the weakest precondition for statement S and postcondition Q.

In our example, `y + 1 > 0` is the weakest precondition. The precondition `y > 100` is not as weak because it allows only a subset of the values accepted by `y + 1 > 0`. The precondition `y = 117` is the strongest of these three assertions because it allows only a single value that was accepted by either of the other two assertions.

Hoare triples

To formalize all this talk about assertions, we introduce something called a Hoare triple, named for Tony Hoare. (Hoare also invented quicksort and many other cool things.) A **Hoare triple**, written `P {S} Q`, consists of a precondition P, a statement S, and a postcondition Q. In a valid Hoare triple, if S is executed in a state where P is true, then Q is guaranteed to be true afterwards. For example:

```
{ x != 0 }        P
y = x*x;          S
{ y > 0 }         Q
```

An aside: {P} S {Q} versus P {S} Q

In other places (including notes from past quarters of 331) you may see curly braces used for statements instead of assertions. Hoare used both conventions in his original paper to mean slightly different things. The difference is subtle, and for the purposes of this course it's just important to pick one convention and stick with it. We chose to put the curly braces around assertions, and you should do the same in this class.

If S is executed in a state where P is false, Q might be true or it might be false; a valid Hoare triple doesn't have to promise anything either way. On the other hand, a Hoare triple is invalid if there can be a scenario where P is *true*, S is executed, and Q is false afterwards. For example, consider the initial state `x = 1, y = -1` for this *invalid* Hoare triple:

```
{ x > 0 }         1 > 0; P is satisfied (note that P says nothing about y)
x = y;            x = -1
{ x > 0 }         -1 < 0; Q is not satisfied. Invalid Hoare triple!
```

To give a subtler example of an invalid Hoare triple:

```
{ x >= 0 }        Invalid Hoare triple
y = 2*x;
{ y > x }
```

Suppose `x = 0` in the initial state. P is satisfied initially, but afterward `y = 0 = x` and Q is not satisfied. If we change Q from `y > x` to `y >= x`, then the Hoare triple becomes valid.

`if/else` statements

So far, we have only looked at sequences of assignment statements executed one after another. We will now consider Hoare triples involving `if/else` statements:

```
{P} if (B) S1 else S2 {Q}
```

When reasoning about `if/else` statements, once again it helps to add an intermediate assertion before/after each line of code. We give the complete structure below, followed by an explanation of each new line:

```
{P}
if (B)
      { P ∧ B }
      S1;
      {Q1}
else
      { P ∧ !B }
      S2;
      {Q2}

{Q1} ∨ {Q2} → {Q}
{Q}
```

What do we know immediately after entering the `if` case containg S1? We know that B is true, or we wouldn't have entered the `if` case. We also know that P is true, because we haven't executed any code that could break it. So, we have the assertion `P ∧ B`. What do we know immediately after entering the `else` case containing S2? Again, P must still be true, and B must be false to have entered the `else` case. So, we have the assertion `P ∧ !B`.

What about Q1 and Q2, and that funny line with the arrow (→)? Q1 and Q2 indicate what's known after S1 or S2 is executed, respectively. Because we always enter one case or the other, we can be sure that Q1 or Q2 will be true after executing the entire `if/else` statement. So to conclude that Q always holds true, we just need to show that Q is true as long as either Q1 or Q2 is true. Written formally, `{Q1} ∨ {Q2} → Q`. If you've never seen this notation before, it is read as "Q1 or Q2 implies Q." It means that if `(Q1 ∨ Q2)` is true, then Q is also true. In other words, if Q1 is true then Q is true, and if Q2 is true then Q is true. (If neither Q1 nor Q2 is true, we don't know anything about Q – it could be true or false.)

Notice that `{Q1} ∨ {Q2} → {Q}` is the second-to-last line in our annotated `if/else` block. This indicates that to prove that Q always holds, we need to actually demonstrate that `(Q1 ∨ Q2) → Q`.

As an example, let's consider writing code to compute the max of two variables x and y and store it in a variable m. We want to prove that the code works correctly. It should work for all inputs, so we have the trivial precondition `{true}`. The postcondition is `{ m = max(x,y) }`, or stated more explicitly, `{ (m=x ∧ x >= y) ∨ (m=y ∧ y >=x) }`. Try writing this code and annotating it with the pattern above to prove that Q always holds.

One possible solution:

```
{true}
if (x > y)
      { true Λ x > y } → { x > y }
      m = x;
      {Q1: m = x Λ x > y }
else
      { true Λ x <= y } → { x <= y }
      m = y;
      {Q2: m = y Λ x <= y }
// {Q1 V Q2} → {Q} trivially
{Q1 V Q2} = { (m = x Λ x > y) V (m = y Λ x <= y) }
      → { m = max(x,y) } = {Q}
```

If {Q1 V Q2} matches {Q} exactly, you can simply write "{Q1 V Q2} → {Q} trivially" above the final assertion containing {Q}. Otherwise, you should write out {Q1 V Q2} (replacing Q1 and Q2 with their actual values) as well as any intermediate steps needed to show how {Q1 V Q2} → Q, as we did above. You want to make your reasoning process as clear as possible to the reader.

Summary so far: Rules for finding the weakest precondition

When we reason about code, we usually want to find the weakest precondition. Even if we're trying to show that our code works in all initial states with no precondition, this can be approached as finding a weakest precondition of {true}. For each type of statement, we need a rule for how to find the weakest precondition.

Assignment statements

We want to find P = wp(x=e, Q). Here, e represents an expression rather than a variable, so it could be replaced with a constant, a variable, a sum of variables … anything that can go on the right-hand side of an assignment statement. As in earlier examples, anything that is true of x after the assignment statement must be true of the value assigned to x beforehand. The weakest precondition P is simply Q with all free occurrences of x replaced by e.

wp(x=e, Q) = Q with all free occurrences of x replaced by e

For example, to find wp(x=y+1, x > 0) we replace x with y+1 in the postcondition x > 0, obtaining the weakest precondition y+1 > 0.

Try the problems below on your own. Starting with the postcondition and statements, fill in the intermediate assertions and weakest precondition:

```
x = x - 2;
z = x + 1;
{ z != 0 }
```

```
x = 2 * y;
z = x + y;
{ z > 0 }
```

```
w = 2 * w;
z = -w;
y = v + 1;
x = min(y, z);
{ x < 0 }
```

The solutions are:

```
{ x !- 1 }
x = x - 2;
{ x != -1 }
z = x + 1;
{ z != 0 }
```

```
{ y > 0 }
x = 2 * y;
{ x + y > 0 }
z = x + y;
{ z > 0 }
```

```
{ v < -1 V w > 0 }
w = 2 * w;
{ v < -1 V w > 0 }
z = -w;
{ v < -1 V z < 0 }
y = v + 1;
{ y < 0 V z < 0 }
x = min(y, z);
{ x < 0 }
```

Statement lists

We want to find the weakest precondition for two consecutive statements, P = wp(S1; S2, Q). It helps to break down the problem by adding an intermediate assertion between S1 and S2, giving:

{P} S1 {X} S2 {Q}

Then we work backwards. We start by finding the weakest precondition for S2 and use this for X, i.e. X = wp(S2, Q). Next, we use X as the postcondition for S1 and find wp(S1, X). The result will be the weakest precondition for the series of statements S1;S2. Replacing X in wp(S1, X), we get:

wp(S1;S2, Q) = wp(S1, wp(S2,Q))

if/else statements

We want to find the weakest precondition for an if/else (conditional) statement wp(IF, Q). As before, we write the statement as

if (B) S1 else S2

Suppose B is true. Because S1 is executed and Q must be true afterward, the weakest precondition for the entire IF statement will be the weakest precondition for S1 and Q, i.e. wp(S1, Q). Analogously, if B is false the weakest precondition will be wp(S2, Q). Putting these two cases together, the weakest precondition for the entire if/else statement is wp(S1,Q) when B is true and wp(S2, Q) when B is false. Written formally:

wp(IF, Q) = (B → wp(S1,Q) ∧ !B → wp(S2,Q))
= (B ∧ wp(S1, Q)) ∨ (!B ∧ wp(S2, Q))

For example, how do we find the weakest precondition for the conditional statement below?

```
if (x < 5)
        x = x*x;
else
        x = x+1;
{ x >= 9 }
```

Using the formula above:

wp(IF, x >= 9) = (x < 5 ∧ wp(x = x*x, x>=9)) ∨ (x >=5 ∧ wp(x = x+1, x >=9))
= (x < 5 ∧ x*x >= 9) ∨ (x >= 5 ∧ x+1 >= 9)
= (x <= -3) ∨ (x >= 3 ∧ x < 5) ∨ (x >= 8)

To practice, find the weakest precondition of the conditional statement below:

```
if (x != 0)
        z = x;
else
        z = x+1;
{ z > 0 }
```

The solution:

wp(IF, z > 0) = (x != 0 ∧ wp(z = x, z > 0)) ∨ (x == 0 ∧ wp(z = x+1, z > 0))
= (x != 0 ∧ x > 0) ∨ (x == 0 ∧ x+1 > 0)
= (x > 0) V (x == 0)
= (x >= 0)

Reasoning about loops

When you hear "loop," you might think immediately of a `for` loop, but we're going to focus on the more fundamental `while` loop:

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

`for` loops, of course, can be rewritten using `while`. The loop `for (`*init*`;` *test*`;` *step*`)` `S` is equivalent to

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

so we lose no generality by restricting our attention to `while`.

When reasoning about a chunk of code involving a loop, there's often an initialization step between the precondition and the start of the loop. The complete code looks more like:

```
{P}
[initialization steps]
while (B)
      S;
{Q}
```

Loops are trickier to reason about than assignment and if/else statements because you don't know how many times the loop body will execute. We have to show that our code is correct if the loop iterates 0 times, 1 times, 2 times, 3 times, 4 times, … . More generally, we need to prove that when the loop terminates, {Q} is satisfied regardless of how many times the loop ran. (We also need to prove that the loop will terminate, but we won't worry too much about that for now.) To do this, we introduce the notion of a **loop invariant**.

A loop invariant, usually written {I}, is a precondition and postcondition for the loop body. {I} must be true at the beginning and end of each iteration of the loop body, though it does not need to hold at every point in the middle of the loop. To show that a given loop invariant is valid, we show (1) that it holds immediately before entering the loop and (2) that if it holds at the beginning of the loop, it also holds at the end. (1) implies that {I} holds at the beginning of the first iteration. (We assume that checking B does not change any variables). By induction, we can then conclude that {I} holds at the beginning and end of every iteration. (Why?) We can also use the fact that {B} is true at the beginning of every iteration or the loop would have terminated.

Since {I} holds true at the end of any iteration, including the final iteration, it must also hold true immediately after the loop exits. We also know {B} is false when the loop terminates or the loop wouldn't have stopped. So to prove that {Q} holds after the loop, we just need to show that {I ∧ !B}→{Q}. (You may have noticed an edge case: what if the loop terminates immediately so the loop body is never executed? Since {I} held before the non-executed loop, it will still hold afterward.)

In summary, to prove that the postcondition is satisfied and our code is correct, we need to show:

1. That {I} holds immediately before the loop, i.e., immediately after the initialization steps.
2. That {I} holds at the end of the loop body, given that {I ∧ B} hold at the beginning of the loop body.
3. That {I ∧ !B}→{Q}

Let's add these assertions to our generalized loop:

```
{P}
[initialization steps]
{I}
while (B)
      {I ∧ B}
      S;
```

```
      {I}
{I ∧ !B} → {Q}
```

Sound easy? The clever part is finding the loop invariant that makes these three conditions true.

As we will see in the examples below, this reasoning process should be performed *as you are writing the code*, shaping what you write. It guides your thought process to help you write clean and correct code on the first try.

When writing loops, you may be tempted to write your code sequentially, starting with {P} and working your way to {Q} one line at a time. It is usually more effective to work from the inside out in roughly the following order:

- Choose a loop invariant and write the loop body (in either order – this is the inventive step).
- Choose B so that {I ∧ !B} → {Q}.
- Add initialization steps to get from {P} to {I}.

This order isn't a hard-and-fast rule, but it's a good place to start.

Example 1

Write a loop to set sum = 1 + 2 + … + n and prove that it is correct.

From the instructions, we have our postcondition: {sum = 1 + 2 + … n}.

Let's start with our loop body. We'll need a counter k. At each iteration, we'll add k to sum and incremen t k. So far, our code looks something like this:

```
{pre: _____________}
[Initialization]
{loop-inv: _____________}
while (B:_____________) {
      {loop-inv: _____________}
      sum = sum + k;
      {_____________________}
      k = k+1;
      {loop-inv: _____________}
}
{loop-inv: _____________}
{post: sum = 1 + 2 + … + n}
```

An aside: notation

Earlier, our preconditions and postconditions looked the same as any other assertions. Now, it will be helpful to attach special labels to our precondition, postcondition, and loop invariant. These labels often make it easier to both read and write proofs by establishing a pattern to follow.

{pre: assertion} denotes a precondition

{post: assertion} denotes a postcondition

{inv: assertion} denotes a loop invariant

What should our loop invariant be? In the loop body, we add k to sum and increment k. This suggests that sum should be 1+2+…+k-1 going into the loop body. Let's call this assertion {I}. If {I} holds at the beginning of the loop, is it guaranteed to hold at the end of the loop? We can find out by writing it as the postcondition for the loop body and reasoning backwards to find the weakest precondition:

```
{sum = 1+2+…+k-1}
sum = sum + k;
{sum = 1+2+…+k}
k = k+1;
{sum = 1+2+…+k-1}
```

Yes, if {I} holds at the beginning of the loop, the fact that we add k to sum and increment sum will ensure that {I} also holds at the end of the loop. We will use {I} for our loop invariant, later adding the initialization steps needed to ensure that it holds on entering the loop for the first time.

Now we need B. Remember that we want the property {I ∧ !B} → {Q}, or

```
{sum = 1+2+…+k-1 ∧ !B} → {sum = 1 + 2 + … + n}.
```

This logic falls neatly into place if we let !B be {k-1 = n}. So then B = !!B is simply

```
{!(k-1 = n)} → {k-1 != n} → {k != n+1}
```

Let's add the new information to our code outline:

```
{pre: _______}
[Initialization]
{loop-inv: sum = 1+2+…+k-1}
while (k != n+1) {
      {loop-inv: sum = 1+2+…+k-1}
      sum = sum + k;
      {sum = 1+2+…+k}
      k = k+1;
      {loop-inv: sum = 1+2+…+k-1}
}
{loop-inv: sum = 1+2+…+k-1}
{sum = 1+2+…+k-1 ∧ k = n+1} → {post: sum = 1 + 2 + … + n}
```

We're almost there! We just need to initialize sum and k to values that will make the loop invariant true initially. One possible initialization is to set k = 1 and sum to a value that establishes that sum is the sum of the elements in the set {1…k-1}. By convention, the set {i…j} for j < i is defined as the empty set, and the sum of an empty set is 0. Because k-1 = 0 < 1, we set sum to 0 to satisfy the loop invariant.

Finally, we just need the weakest precondition for the entire sequence of code. We need k <= n+1 initially or the loop will never terminate. Reasoning backwards, we arrive at the precondition n >= 0.

The complete code, with assertions, looks like this:

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

Example 2

Write a loop to set max = largest value in items[0…size-1]

We get our postcondition directly from the problem:

```
{max = largest value in items[0…size-1]}
```

A simple approach is to look through `items` one element at a time, keeping track of the largest value seen so far. The loop terminates when we've examined all elements. Much like in a typical `for` loop, we'll keep a counter k, examining `items[k]` during each iteration of the loop and updating `max` if needed. From this description, the loop invariant {I} emerges naturally: {`max = largest in items[0…k-1]`}.

To choose {B}, recall that we want {`I ∧ !B`} → {Q}, or

```
{max = largest in items[0…k-1] ∧  !B}

→ {max = largest in items[0…size-1]}.
```

This works if we set {`!B`} to {`k = size`}, or {B} = {`k != size`}.

Finally, we make {I} true initially by setting `k = 1` and `max = items[0]`. This assumes the list is non-empty, so our precondition is `size > 0`.

Let's put this all together:

```
{pre: size > 0}
k = 1;
max = items[0];
{loop-inv: max = largest in items[0…k-1]}
while (k != size) {
      {loop-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 to do
            {max = largest in items[0…k]}
      }
      k = k+1;
      {loop-inv: max = largest in items[0…k-1]}

}
{loop-inv: max = largest in items[0…k-1]}
{max = largest in items[0…k-1] ∧ k = size}
      → {max = largest in items[0…size-1]}
```

An aside: the code works, but only for non-empty lists (size > 0). What if the list is empty? We could throw an exception, but we might like to accept the broadest set of inputs possible (i.e., make the precondition as weak as possible). One way we could do that is to return the result `Integer.MIN_VALUE` if `size == 0`. (Is this a good design decision? It depends on what properties we want to hold for this method.) We end up with something like:

```
// return largest value in list if not empty
// otherwise, return Integer.MIN_VALUE
public int max() {
      if (size == 0) {
            return Integer.MIN_VALUE;
      } else {
            // code from above
      }
}
```

Example 3

Given a[0…n-1], reverse the elements in a.

Choosing the loop invariant is a little subtler than for previous problems. We need to say something about the state of the whole array at each iteration. Let's start with our precondition and postcondition to establish the initial and final state of the array:

An aside: notation for historical values

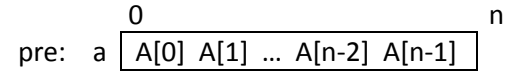
Sometimes an assertion needs to refer to the earlier value of a variable. To prove that

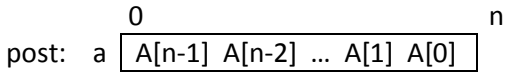
```
t = x; x = y; y = t;
```

swaps x and y, the postcondition must refer to their initial values. You can do so with subscripts or case distinctions, e.g.:

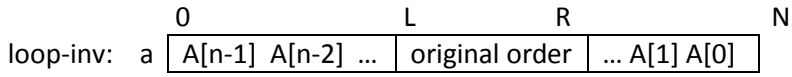
{pre: $x=x_{pre} \wedge y=y_{pre}$}

{post: $x=y_{pre} \wedge y=x_{pre}$}





In our loop invariant, we will divide the array into three regions: two regions at the front and back that have already been swapped and a region in the middle that still needs to be swapped.



At each step of the loop, we will swap the items at the beginning and end of the unswapped section and adjust the endpoints. Let us define L and R as the first and last indexes of the unswapped section, respectively. Alternatively, L could be the index right before the unswapped section and/or R could be the index right afterward. It doesn't really matter which convention you use (though one might result in cleaner code) – just pick one and *use it consistently!* Bugs happen when someone forgets what they chose and writes part of the code assuming a different convention.

When the unswapped section only contains 0 or 1 elements, the entire list has been reversed. This occurs when L >= R, so {B} = {L < R}. Initially the entire list is unswapped, and we initialize L and R accordingly.

Putting it all together:

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

For the purposes of this example, we assume we have a swap() procedure that we know is correct. This is simply a convenience so we can concentrate on reasoning about the loop. (Astute readers may notice that swap() as currently defined could never work correctly as a Java method. It is simple enough, however, to replace swap() with three inline assignment statements anywhere it occurs. Making this replacement and proving it correct is left as an exercise for the reader. ☺)

Example 4

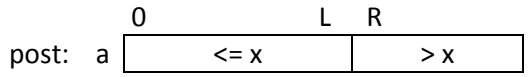
Binary search. Given a value x and sorted array a[0…n-1], find the index of x, if present. If x is not present, return an appropriate location in the array that can be used to insert x if desired.

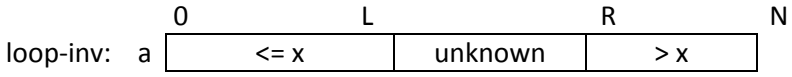
The basic idea of binary search is to maintain three regions of the array: a region at the front where all values are <= x, a region at the end where all values are > x, and an "unknown" region in the middle. At each iteration of the loop, we compare x to the element at the center of the unknown region and adjust the endpoints of the region accordingly. We repeat this process until the unknown region is empty. Then x will be located or inserted at the end of the first region.

We will use L and R to represent the indexes immediately before and after the unknown region. Notice that this is a different convention than we used in the previous example, where L and R were the first and last elements of the middle region. Again, it doesn't matter which convention we use as long as we are consistent, except that the one choice might make the code simpler or easier to follow than the other.

Our precondition is that the loop is already sorted. Our loop invariant shows what we know about the array at each step along the way, and our postcondition shows what we know at the end:

pre: a[0] <= a[1] <= … <= a[n-1]





We can also write our loop invariant as:

```
loop-inv: a[0…L] <= x ∧ a[R…n-1] > x ∧ a[L+1…R-1] unknown
```

The loop terminates when the unknown region is empty, i.e. L+1=R. Initially the entire list is unknown, and we initialize L and R accordingly.

Putting it all together:

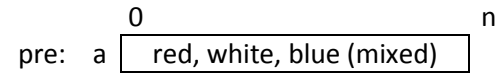
```
L = -1;
R = n;
while (L+1 != R) {
      mid = (L+R)/2;
      if (a[mid] <= x)
            L = mid;
      else  // a[mid] > x
            R = mid;
}

// x is found if L >= 0 &&  a[L] = x (note that the short-circuit
// property of && is essential here)
```

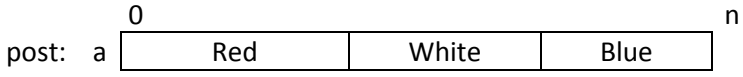
As in the previous example, we glossed over proving the body of the loop in detail.

Example 5

("Dutch national flag" problem) 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.

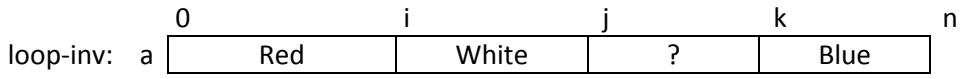
Once again, we describe the state of the array in the precondition, postcondition, and loop invariant. The precondition describes an array with red, white, and blue elements of unknown quantities in an unknown order. The postcondition is a sorted array.





We can see that the array naturally has three regions in the postcondition. For the loop invariant, we add an "unknown" region representing the values not yet sorted. We keep track of the endpoints of the four regions. At each step, we take a value from the unknown region, put it in the appropriate sorted region, and adjust the endpoints of all the regions accordingly.

Where should the unsorted region go? At the end? Somewhere in the middle? We choose to keep it between the white and blue regions, although it could just as easily have been between the red and white areas. Both of these choices have the advantage that pebbles that have been moved to the red and blue regions do not need to be moved further as the algorithm progresses. Our loop invariant is then:



a[0…i-1] red ∧ a [i…j-1] white ∧ a[j…k-1] unknown ∧ a[k…n-1] blue

Our code is then straightforward:

```
i = 0; j = 0; k = n;
while (j+1 != k) {
      if (a[j] == white) {
            j = j+1;
      } else if (a[j] == blue) {
            swap(a[j], a[k-1]);
            k = k-1;
      } else {  // a[j] is red
            swap(a[i], a[j]);
            i = i+1;
            j = j+1;
      }
}
```

As before, we use the swap() function to interchange array elements.