# Course 141
# Reasoning about Programs – Part III

Krysia Broda

## Using Invariants to Construct Algorithms

- **Binary Chop**
- **Restoring Flags**
- **Using Haskell to guide reasoning**
- **Path-finding**

## Binary Chop

- This is a very useful algorithm,
  but one that is difficult to get right first time.
- It illustrates the problem solving method of **Divide and Conquer**:
  given a big problem —
  **divide** it into smaller parts;
  solve each part separately (easier than the original)
  hence **conquer** the original problem.
- For binary chop in particular, the key to making good use of the strategy is to know exactly what you are trying to do.
- Illustrates reasoning with pre/post conditions and invariants.

## WHAT IS BINARY CHOP ABOUT?

**Problem:** Given a *sorted* array a [say of **int**] and an **int** x as input, find whereabouts in a the element x occurs.

If a wasn't sorted, there'd be little alternative to inspecting all the elements of a one by one until x is found.

BUT – for a sorted array we can be smarter ☺

**Rough idea** (assuming a is sorted in ascending order):

Look at the element half way along a.
If this is bigger than x, then x must be in the first half.
If it is smaller, then x must be in the second half.
Either way, we have cut the search area by a factor of 2.
Repeat this until x is found.

## SPECIFICATION – FIRST ATTEMPT

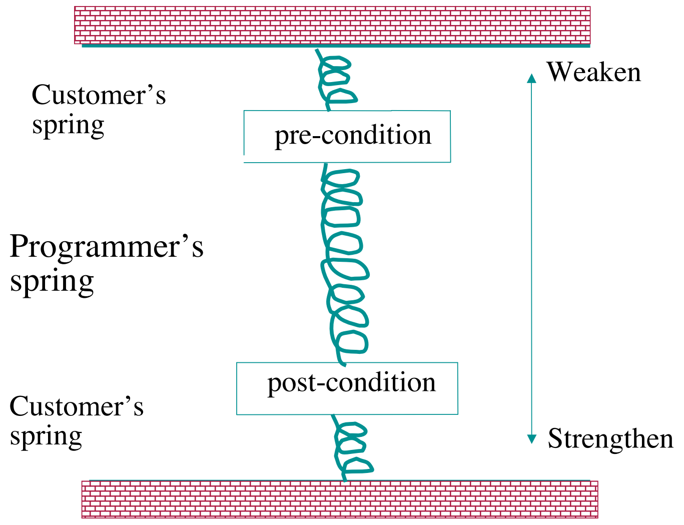
```
int search( int [] a, int x) {
// Pre:     Sorted(a)
// Post:    a[r]= x
  }
```

Recall:

- in Post **r** denotes the value returned by the function

**Is that it?**
**Things we might consider:**

1. (AE)
2. (SA)
3. (DU)
4. (NI)



## FIRST PROBLEM – *SUPPOSE X IS NOT IN THE ARRAY* ?

*What answer would we like?*

One possible use for search is to find whereabouts in a to insert a new element x so that a remains sorted. We look for the boundary between the elements < x and those > x.

There are two ways of describing this boundary by **r**.

| | | | |
|---|---|---|---|
| Way 1: | a [**r**] < x | a [**r**+1] > x | x goes at a[**r**+1] |

Look at the boundary cases:

| | all elements of a are > x | all elements of a are < x |
|---|---|---|
| Way 1 | **r** is -1 (a[0] > x) | **r** is a.length -1 |

**OR**:

| | | | |
|---|---|---|---|
| Way 2: | a[**r**-1] < x | a [**r**] > x | x goes at a[**r**] |

Look at the boundary cases:

| | all elements of a are > x | all elements of a are < x |
|---|---|---|
| Way 2 | **r** is 0 | **r** is a.length (a[a.length-1]<x) |

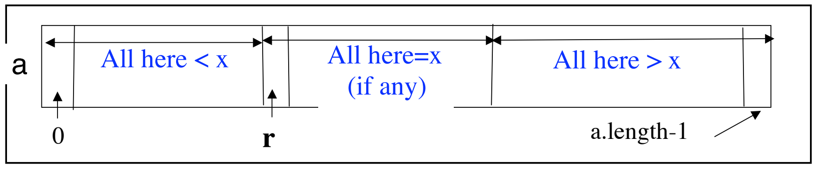
**Way 2** is our standard. Then **r** is *the smallest index where the array element is* > x (or a.length if all the elements are < x).

## NEXT PROBLEM – *WHAT IF X OCCURS MORE THAN ONCE IN THE ARRAY?*

- Because a is ordered, all the occurrences will be together.
- Would we like **r** to be the index of the first or the last?
- *Choose the first*, so that **r** is the smallest index.
- **r** defines the boundary between the elements < x and those ≥ x.
- This matches our choice for when x doesn't occur at all.

So in all possible cases…

**r** is the smallest index where the array element is ≥ x, or a.length if all the elements are < x.



## SPECIFICATION – FINAL ATTEMPT

```
int search(int [] a, int x) {

// Pre:    Sorted(a)  i.e.
//         ∀i,j:int ( 0 ≤ i ≤ j< a.length →a[i]≤a[j])

// Post: r is the smallest index where the element is ≥ x,
//         or a.length if all the elements are < x

//         i.e.  0 ≤ r ≤ a.length
//         ∧ ∀i.int (0 ≤ i < r → a[i]< x)
//         ∧ ∀i.int (r ≤ i < a.length → a[i] ≥ x)
//         ∧ a= a0 ∧ x=x0 (ie a and x are unchanged)
```

## LOOP INVARIANT (REMINDER)

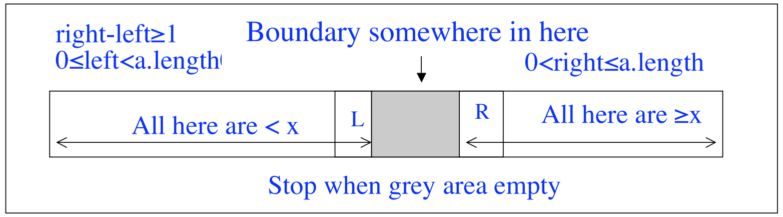
```
//start
    |  //Loop Initialisation code
    ==> loop invariant true (1st time)
//while loop   --<--|  Loop invariant true again
|       |           |     (2nd, 3rd, .... times)
|       |           |  "OK so far and
|       |           |  Make progress to postcondition"
|       |---->------
 // loop end (Loop invariant still true here)
// Loop Finalisation code   (Ensures postcondition)
```

***Assume variant: v:int.***

***Loop terminates if v1 > v2 >.... > vk > ... > I in the loop.***
***where vi=value of v inside loop for ith time, and***
***I is a fixed int, often I = 0***

## DIAGRAM (ILLUSTRATES LOOP INVARIANT)

Keep two variables, left and right, to show how far we've narrowed the search area. Boundary must be between left and right.



The shaded region goes from left+1 to right-1 *inclusive*.

**Test intuition:**

(i) a[left] <x and a[right]>=x - therefore left <right – **why**?

(ii) If left+1=right and x is in a,
then right<a.length and a[right]=x – **why?**
**i.e. result really is the smallest index of x in a.**

***Fairly informal proofs of properties (i) and (ii) on slide 11 – consequences of the invariant***

**Proof of Property (i)**.

Assume the invariant properties (1) - (4) for some arbitrary values of left and right:
(1) 0≤left<a.length
(2) 0<right≤a.length
(3) ∀i(0≤i≤left→a[i]<x)
(4) ∀i(right≤i<a.length→a[i]≥x)
We are required to show (RTS) that left<right.
We can use proof by contradiction: Suppose left≥right (*).
Case 1: If right=a.length then by assumption left≥a.length, which contradicts (1).
Case 2: If right<a.length, then by (1) and assumption (*) right≤left<a.length.
Hence by (4) a[left]≥x and by (3) a[left]<x, a contradiction.

**Proof of Property (ii)**

Assume invariant properties (3) and (4) and givens (5) and (6):
(3) ∀i(0≤i≤left→a[i]<x)
(4) ∀i(right≤i<a.length→a[i]≥x)
(5) 0<left+1=right≤a.length
(6) x is in a
RTS (a) right<a.length and (b) a[right]=x.

(a) Suppose for contradiction that right=a.length.
From (5) left=a.length-1 and by (3) all elements in a are <x which contradicts x is in a.

(b) Assume result (a).
Suppose for contradiction that a[right]≠x.
From (4) a[right]≥x, hence a[right]>x; since right=left+1 we have (7) a[left+1]>x.
By (3) we have (8) a[left]<x.
By sortedness, ∀i(0≤i≤left→a[i]≤a[left]) and ∀i(a.length>i≥left+1→a[i]≥a[left+1]).
Together with (7) and (8), we deduce ∀i(a.length>i≥left+1→a[i]>x) and
∀i(0≤i≤left→a[i]<x).
Therefore, for no i does a[i]=x, contradicting that x is in a.

**Exercise**: Write the proofs out using ND and the definition of sortedness on Slide 9.

For property (ii) "x is in a" can be written as ∃j(0≤j<a.length ∧ a[j]=x).
Use Pandora if you like. But you may need to use such obvious properties as
"if 0≤i<a.length and 0≤left<a.length then either 0≤i<left or left≤i<a.length", or
x<y→x≠y and x>y→x≠y.
That's why we call the proofs "informal".

## TOWARDS THE CODE

**Loop invariant:**

(I1) 0 ≤ left < right ≤ a.length ∧ a=a0 ∧ x=x0
(I2) ∀i.**int** (0 ≤ i ≤ left → a[i] < x) ∧
(I3) ∀i.**int** (right ≤ i < a.length → a[i] ≥ x)

**Loop variant:** right - (left+1) = right - left - 1 (this is a <u>number</u>)
(i.e. the size of the uncharted area in the middle)

```
while
  (right - left > 1) {   // variant >0
  :                      // re-establish invariant
  :                      // and make search area smaller
  }
return right;            //r= right
```

**Loop Initialisation** (establishing the invariant)
(We assume a=a0 and x=x0 from now on)

‡ First try:

```
right = a.length;   left = 0;
// invariant:0≤left<right≤a.length ∧
//           ∀i:int(0≤i≤left → a[i]<x) ∧
//           ∀i:int(right≤i<a.length → a[i]≥x)
```

***Wrong* – Does not always establish invariant**

***Why*? – there are 2 errors; consider some special cases.**

‡ Second try:

```
right = a.length;
if (a.length==0 || a[0] >= x) return 0; else left = 0;
// mid: a.length>0 ∧ left=0 ∧ a[0]=a[left]<x ∧ left<right ∧
// invariant: (I1)  0≤left<right≤a.length ∧
//            (I2)  ∀i:int(0≤i≤left → a[i]<x) ∧
//            (I3)  ∀i:int(right≤i<a.length → a[i]≥x)
```

**Proof** invariant is established just before while loop (a≠[] and a[0]<x)
(I1): RTS (Required To Show)
0≤0<a.length≤a.length (ie substitute for values of left etc.)
True by arithmetic and assumption that here a is non-empty

(I2): if i=0 then a[i] = a[0]<x and 0≤i≤0 so implication true.
For all other i the condition of implication is false.

(I3): There is no i such that right ≤ i < a.length (right=a.length)
So condition of implication always false

We can write the proof a bit more formally as follows. Let left1, right1 etc. be values of left and right after executing the else statement.

*Given*: After the else condition of the if-statement, we know (as a mid-condition)
(M1) left1=0 (M2) a.length>0
(M3) right1=a.length (M4) a[0]=a[left1]<x (note a[0] is defined as a is non-empty).

Since there are no further changes before the while loop, left1 etc. are also the values just before the while loop. Hence we want to show the Invariant holds for left1, right1 etc.

*To show (I1)*: $0\le left1<right1\le a.length \iff 0\le 0<a.length\le a.length$ (Use M1 and M3).
$0\le 0$ and $a.length\le a.length$ by arithmetic. $0<a.length$ is given.

*To show (I2)*: $\forall i(0\le i\le left1\rightarrow a[i]<x) \iff \forall i(0\le i\le 0\rightarrow a[i]<x)$ (Use M1).
Let I be an arbitrary int. We show $0\le I\le 0\rightarrow a[I]<x$
That is, assume $0\le I\le 0 \iff I=0$ and show $a[I]<x$.
$a[I]=a[0]<x$ (by M4).

*To show (I3)*:
$\forall i(right1\le i<a.length\rightarrow a[i]\ge x) \iff \forall i(a.length\le i<a.length\rightarrow a[i]\ge x)$ (by M2)
Let I be an arbitrary int. We show $a.length\le I<a.length\rightarrow a[I]\ge x$.
Assume $a.length\le I<a.length \iff \perp \implies a[I]\ge x$ ($\perp$E).

*Question*: In the proofs of (I2) and (I3) what ND rules were used at the outer level?
*Answer*: The $\forall$I rule followed by the $\rightarrow$I rule.

**Loop Finalization** (establishing Post)

//Post: (P1) $0 \le \mathbf{r} \le a.length \wedge$
// (P2) $\forall i.\mathbf{int}\ (0 \le i < \mathbf{r} \rightarrow a[i]< x) \wedge$
// (P3) $\forall i.\mathbf{int}\ (\mathbf{r} \le i < a.length \rightarrow a[i] \ge x)$

There are two cases -

either exit before while loop starts or exit after while loop ends

**Case 1 (exit before loop) a.length=0 or (a.length>0 ∧ a[0]≥x) r=0**

(Informally–if the "if" condition in the code=true then **r**=0 is correct)
(P1) $\iff 0\le 0\le a.length \iff$ true (arithmetic and Givens)
(P2)$\iff\forall i.\mathbf{int}(0\le i<0\rightarrow a[i]<x)\iff$true: for all i, $0\le i<0$ is false
(P3) $\iff \forall i.\mathbf{int}\ (0 \le i < a.length \rightarrow a[i] \ge x)$.

There are 2 sub-cases for (P3):
(i) *a.length=0* ==> $0\le i<0$ is false for every i==>implication true
(ii) *a.length>0*: $a[0]\ge x$ and a is sorted ==>$x\le a[0]\le a[i]$ for every i.

**Loop Finalization** (establishing Post) continued

//Post: (P1) $0 \le \mathbf{r} \le a.length \wedge$
// (P2) $\forall i.\mathbf{int}\ (0 \le i < \mathbf{r} \rightarrow a[i]< x) \wedge$
// (P3) $\forall i.\mathbf{int}\ (\mathbf{r} \le i < a.length \rightarrow a[i] \ge x)$

**Case 2 (exit after loop) Let left2 be left's value just after loop, etc.**

right2-left2≤1(Loop test failed)

(I1) 0≤left2<right2≤a.length

**r**=right2 (return statement)

(Informally – if left becomes right-1, then **r** = right is correct.)

right2-left2≤1 ==> right2≤left2+1 and by (I1) right2≥left2+1

∴ right2=left2+1.

Substitute **r** for right2 and **r-1** for left2 in (I2) and (I3) – gives Post.
eg in (I2) $\forall i.\mathbf{int}\ (0 \le i \le left2 \rightarrow a[i] < x)$ put **r-1** for left2, then
$\forall i.\mathbf{int}\ (0 \le i \le \mathbf{r-1} \rightarrow a[i] < x) \iff \forall i.\mathbf{int}\ (0 \le i < \mathbf{r} \rightarrow a[i] < x)$ (P2)

## NOTE ON NAMING CONVENTIONS IN PROOFS

It is convenient to label values of variables at particular points in a program to clarify reasoning about them.

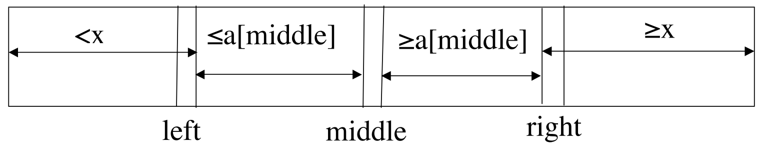
Typical points could be
just before a loop starts
just after a loop is entered
just before a loop ends
just after a loop exits
at particular points in a code sequence

eg earlier we used left1, right1 for values of left and right before entering the while loop and left2, right2 for the values after exiting the while loop. We'll see some more examples soon.

The names we choose are arbitrary. eg we could have chosen l1, r1
Make sure you state what names are used, and what they stand for.

## RE-ESTABLISHING THE INVARIANT (1)

The idea is to define ‘middle’ as (left+right) / 2.



if a[middle]≥x then right=middle
 i.e. ∀i.**int** (right ≤ i <a.length → a[i] ≥ x) **WHY?**

if a[middle<x then left = middle
 i.e. ∀i.**int** (0 ≤ i ≤left → a[i] < x) **WHY?**

## RE-ESTABLISHING THE INVARIANT (2)

The idea is to define ‘middle’ as (left+right) / 2.

If a[middle] < x, then we can replace left by middle.
Then if i ≤ left we have a[i]≤a[left]=a[middle]< x.

Otherwise, if a[middle] ≥ x, then we can replace right by middle.
For if i ≥ right (= middle), then a[i] ≥ a[right]=a[middle] ≥ x.

```
middle = (left+right) / 2;
if (a[middle]< x) left = middle;
else right = middle;
```

Within the loop the while condition is true and we can show
right-left >1 ==> right-left≥2 ==> left<middle<right
(uses fact about integer division - see later)

## RE-ESTABLISHING THE INVARIANT (3)

After setting middle to left or to right (as in the code),
three things need to be proved that are slightly *delicate*:

a) 0 ≤ middle < a.length, so that a[middle] is defined;

b) that we *still have* 0 ≤ left < right ≤ a.length,
so that the invariant (1st part) has been re-established;

(We will show the second part of the invariant in a while.)

c) that the variant, right–left-1, has strictly decreased and is ≥0

Given the invariant is true at the start of the loop they all follow from the following fact, which could usefully be included as a comment:

$\boxed{\text{if left} \le \text{right-2, then left} < \text{middle} < \text{right}}$

**<u>Exercise:</u> Show formally that the Loop Finalization implies Post and that the properties a), b), c) hold.**

We can write the proof that the loop finalisation implies Post a bit more formally as follows.

*Given:* a is sorted (G), RTS (P1), (P2) and (P3) of Slide 18.

*Case 1*: (exit before loop starts) **r**=0=a.length.
(P1) <==> 0≤**r**≤a.length<==>0≤0≤0 <==>True by arithmetic
(P2) <==> ∀i(0≤i<**r**→ a[i]<x)<==>∀i(0≤i<0→ a[i]<x)
 <==>True since for all i, 0≤i<0 is False.
(P3) <==> ∀i(**r**≤i<a.length→ a[i]≥x)<==>∀i(0≤i<0→ a[i]≥x)
 <==>True since for all i, 0≤i<0 is False

*Case 2*: (exit before loop starts) a.length>0, a[0]≥x and **r**=0.
(P1) <==> 0≤**r**≤a.length<==>0≤0≤a.length
 <==>True by arithmetic and the case assumptions
(P2) <==> ∀i(0≤i<**r**→ a[i]<x)<==>∀i(0≤i<0→ a[i]<x)
 <==>True since for all i, 0≤i<0 is False.
(P3) <==> ∀i(**r**≤i<a.length→ a[i]≥x)<==>∀i(0≤i<a.length→ a[i]≥x)
 <==>True since for all i a[i]≥a[0]≥x (by (G) and case).

*Case 3*: (exit after loop ends) Let values of variables be right2, left2, etc.
 right2-left2≤1 (loop exit) and **r**=right2.

*Given Invariant after the loop*:
(I1) 0≤left2<right2≤a.length
(I2) ∀i(0≤i≤left2→ a[i]<x)
(I3) ∀i(right2≤i<a.length→a[i]≥x)

Note: $right2-left2\le 1 \Longleftrightarrow right2\le left2+1$ and $left2<right2$ (by I1) $\Longleftrightarrow left2+1\le right2$;

$\therefore\ left2+1=right$

(P1) $\Longleftrightarrow 0\le \mathbf{r}\le a.length \Longleftrightarrow 0\le right2\le a.length \Longleftrightarrow 0\le left2+1\le a.length.$

To show $0\le left2+1$: from (I1), $0\le left2 \Longrightarrow 0\le left2+1$

To show $left2+1\le a.length$: from (I1) $left2 < a.length \Longleftrightarrow left2+1\le a.length$

(P2) $\Longleftrightarrow \forall i(0\le i<\mathbf{r}\rightarrow a[i]<x) \Longleftrightarrow \forall i(0\le i<right2\rightarrow a[i]<x) \Longleftrightarrow \forall i(0\le i<left2-1\rightarrow a[i]<x)$

$\Longleftrightarrow \forall i(0\le i\le left2\rightarrow a[i]<x) \Longleftrightarrow$ True by (I2).

(P3) $\Longleftrightarrow \forall i(\mathbf{r}\le i<a.length\rightarrow a[i]\ge x) \Longleftrightarrow \forall i(right2\le i<a.length\rightarrow a[i]\ge x)$

$\Longleftrightarrow$ True by (I3).

To show the Properties a), b), c) hold after the code on slide 22 we first show, for any values of left and right, and where middle = (left+right)/2, that

$left\le right-2 \rightarrow left<middle<right;$

Suppose $left\le right-2$. RTS $left<middle$ and $middle<right$.

$left+2\le right \Longrightarrow middle \ge (left+left+2)/2=left+1 \Longleftrightarrow middle>left$, and

$middle \le (right-2+right)/2=right-1 \Longleftrightarrow middle <right.$

Now we can show properties a), b), c)

Let left3 and right3 be the values of left/right before the reassignment of left or right and left4, right4 the values after the reassignment. Let middle4 be value of middle.

*Given*:

(G1) $middle4 = (left3+right3)/2$; (Assignment)

(G2) $left3\le right3 - 2 \rightarrow left3<middle4<right3$;

(G2) was proved above - holds for left3 and right3 since before assignment

(G3) $0\le left3<right3\le a.length$ (from Invariant (I1) - before assignment)

(G4) $left3\le right3 - 2$ (loop test);

(G5) $left3<middle4<right3$ (from (G2) and (G4))

a) $\Longleftrightarrow 0\le middle4<a.length.$

From (G3) and (G5) obtain (G6) $0\le left3<middle4 <right3\le a.length$

$\Longrightarrow 0\le middle4<a.length$

There are two cases for each of (b) and (c):

Case (i): left4=middle4 and right4=right3

Case (ii) left4=left3 and right4=middle4.

(b) Case (i): RTS $0\le left4<right4\le a.length \Longleftrightarrow 0\le middle4<right3\le a.length.$
Follows from (G6).

Case (ii): RTS $0\le left4<right4\le a.length \Longleftrightarrow 0\le left3<middle4\le a.length.$
Follows from (G6).

c) Notice that from (b) $right4-left4-1\ge 0$, so variant$\ge 0$.

Variant before loop code = right3 - left3 - 1. Variant after loop code = right4 - left4 - 1.

Case (i) $right4 - left4 - 1 = right3-middle4-1<right3-left3-1$ (from (G6))

Case (ii) $right4 - left4 - 1 = middle4-left3-1<right3-left3-1$ (from (G6))

So in both cases variant$\ge 0$ and variant decreases.

## HOW DO WE KNOW THAT LEFT < MIDDLE < RIGHT?

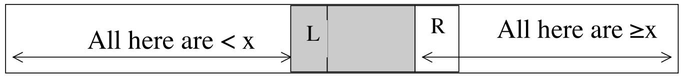
Remember we are assuming $left \le right-2$. Then

$$middle = (left+right) / 2 \ge (left+left+2) / 2 = left+1$$

$$middle = (left+right) / 2 \le (right+right-2) / 2 = right-1$$

This depends on the facts $(n+n-2) / 2 = n-1$ and $(n+n+2) / 2=n+1$.

This solves a problem that might arise if left is taken as the first unchecked element and looping continues until left>=right. For then, computing middle when left=right-1 might give left or it might give right, depending on the exact definition of integer division in the language. And then the program could go wrong. (**See Exercises**).



A slightly different version of the algorithm is obtained if left is taken to be the first element of the unexplored region instead of the last element of the region of elements known to be <x, as was done in the given algorithm. In this alternative, the first part of the invariant must be changed from left<right to left≤right. The loop initialisation is also slightly easier, as there is no need for the "if-statement" and both (I2) and (I3) are vacuously true just before the loop. The while condition in this case would be right-left>0. When false, together with the invariant property left ≤right, this gives left=right, which means, in effect, that there is no unexplored territory, so right can be returned as the result. However, the code to determine middle is rather more delicate:

After computing middle and checking a[middle], if it is too large then right = middle. But if it is too small then left = middle+1. The delicate bit is to show the variant still decreases. For this, we need to be sure that middle<right. If middle = (left+right) / 2, can we show that, if left<right (while condition true), then middle<right? It will depend on how integer division is treated in the language. The difficult case is when left = right-1. If care isn't taken the program could loop for ever. The Problem sheet exercises guide you to fill in the details.

For yet another version (see Problem sheet) if a 3-way test is available, then it appears that when a[middle]=x the loop exit could be made early. For example, suppose the very first time the loop is executed gives rise to a[middle]=x. Is it right to return with **r**=middle? NO!

**Exercise**. Explain why you could not guarantee the given postcondition (without some additional computation). Give a new postcondition that can be guaranteed.

Even if the original postcondition is kept, the 3-way test may still be useful:
**Exercise.** Assume that left and right indicate the first and last elements of the *uncharted* territory and a 3-way test is available. Make appropriate changes to invariant and code.

## THE CODE FOR BINARY CHOP

```
int search (int [] a, int x) {
// Pre:  Sorted(a)
int left, middle;
int right =a.length;
if ((a.length==0)||(a[0]>=x)) return 0; else left = 0;
//a[left]<x
while
    // Loop invariant : see slide 14
    // Loop variant = right – left-1
   (right-left>1) {
    middle = (left+right) / 2;     // left < middle < right
    if (a[middle]< x) left = middle; else right = middle;
  }
  return right;
}
```

## SUMMARY OF PROOFS SO FAR

We proved that the post-condition follows wherever the code exits; (slides 18,19, 24 and 25) ☺

We proved that part of the invariant is re-established (slides 24, 25, 26 and 27) ☺

We proved the initialisation established the invariant at the start of the while loop (slides 16 and 17) ☺

We still have to show the invariant is re-established (next lecture) ☹

We showed the variant decreased (slide 26) ☺

We now show all array accesses are valid:

In **if** ((a.length==0)**||**(a[0]>=x)) // **a[0] is valid because:**

In middle = (left+right) **/** 2;  // left < middle < right and (I1)
  **if** (a[middle]< x)  // **a[middle] is valid because:**

## DOCUMENTATION

- All serious programs have to be “documented” – i.e. there has to be a written explanation of what they do and how they [are supposed to] work. This is usually incorporated as comments.
- The comments in search should show the level of detail that is most useful – not a full formal proof, but must show the most important steps.
- If a formal correctness proof is required, the comments indicate how it would be constructed.
- Even if no (extra) comments, the *loop invariant* gives a solid framework in which to understand the working of the program.
- If there’s a suspicion of a mistake, or if someone else is trying to understand your code, the framework immediately suggests specific questions: e.g. *Does* the loop body re-establish the invariant? *Is* the variant decreased each time? *Are* array accesses OK?

## AN INTERLUDE ....

We can run search and see how many comparisons it takes.
Theory says it is $\log_2$size, where the array has size elements.
Let's see ...
Set up arrays of varying lengths with increasing random integers.
Then search for various integers.

## Reasoning from Specifications

- We revisit binchop and complete the reasoning by showing the invariant is maintained.
- We look at variations of binchop, including changes to the specification and changes to the code.
- We reason using the postcondition to show properties of binchop.
- **Mostly revision of things learned recently applied to binchop.**

## VARIATIONS OF BINARY CHOP CODE/SPECIFICATION

We'll consider two variations (there are several other possibilities too).

Variation 1: we change the specification (pre/post conditions)

Variation 2: we change the code

Each time we must reconsider the proof;

- in the code change we also need to consider whether to adapt the specification,
- and in the specification change we need to consider whether to change the code.

## VARIATION 1: CHANGE OF PRE/POST

Suppose the precondition includes the fact that x is in a.

$\wedge\ \exists y{:}\mathrm{Nat}\ (y<a.\mathrm{length} \wedge a[y]=x)$

We can then strengthen the postcondition, which was:

$$0\le \mathbf{r}\le a.\mathrm{length}\ \wedge$$
$$\forall i.\mathbf{int}(0\le i<\mathbf{r}\rightarrow a[i]<x) \wedge \forall i.\mathbf{int}(\mathbf{r}\le i<a.\mathrm{length}\rightarrow a[i]\ge x)$$

to

$$0\le \mathbf{r}<a.\mathrm{length} \wedge a[\mathbf{r}]=x \wedge \forall i.\mathbf{int}(0\le i<\mathbf{r} \rightarrow a[i]<x)$$

In other words, the result **r** is the index of the first occurrence of x in a.

**Question:**
**Why does the new-Post follow from the old-Post and new-Pre?**
**(Assume no change in code)**

## VARIATION 1 (CONTINUED)

New-Precondition: a is sorted and x is in a.

(Pre1) $\forall i,j{:}\mathrm{int}\ (0 \le i \le j< a.\mathrm{length} \rightarrow a[i]\le a[j]) \wedge$

(Pre2) $\exists y{:}\mathrm{Nat}\ (y<a.\mathrm{length} \wedge a[y]=x)$

Old-Post: (Post3) $0\le\mathbf{r}\le a.\mathrm{length}\ \wedge$
$\forall i.\mathbf{int}(0\le i<\mathbf{r}\rightarrow a[i]<x) \wedge \forall i.\mathbf{int}(\mathbf{r}\le i<a.\mathrm{length}\rightarrow a[i]\ge x)$

==> new-Post:(Post4) $0\le\mathbf{r}<a.\mathrm{length} \wedge a[\mathbf{r}]=x \wedge \forall i.\mathbf{int}(0\le i<\mathbf{r} \rightarrow a[i]<x)$

We use proof by contradiction.

*Case 1: $0\le\mathbf{r}<a.\mathrm{length}$*

From (Post3) $a[\mathbf{r}]\ge x$ and $\forall i.\mathbf{int}(0\le i<\mathbf{r}\rightarrow a[i]<x)$ (*)

Assume for contradiction that $a[\mathbf{r}]>x$;

By (Pre1) $\forall i.\mathbf{int}(\mathbf{r}\le i<a.\mathrm{length}\rightarrow a[i]\ge a[\mathbf{r}]>x)$ (**)

Hence $\forall i.\mathbf{int}(0\le i<a.\mathrm{length}\rightarrow(a[i]<x \vee a[i]>x)$ (from (*) and (**))

This contradicts (Pre2). Therefore $a[\mathbf{r}]=x$

*Case 2: $r=a.\mathrm{length}$*. Then (Post3) contradicts (Pre2) so case impossible

## VARIATION 2: CHANGE OF CODE

Suppose we use a case test on a[middle] with one of 3 outcomes:

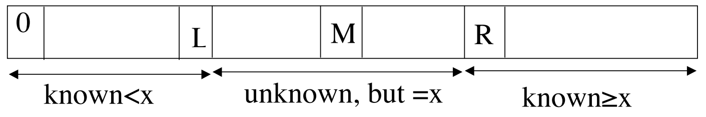
a[middle]<x, a[middle]=x or a[middle]>x.

If a[middle]=x, suppose we write return middle.

The original postcondition was

$$0 \le \mathbf{r} \le a.\text{length} \wedge$$
$$\forall i.\mathbf{int}(0 \le i < \mathbf{r} \rightarrow a[i] < x) \wedge \forall i.\mathbf{int}(\mathbf{r} \le i < a.\text{length} \rightarrow a[i] \ge x)$$

**Q: Why is this <u>not</u> now guaranteed always to be true?**
**HINT: Consider an array in which there are several values of x.**



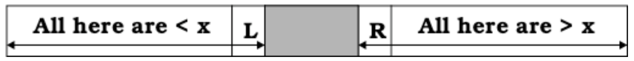
## VARIATION 2 (CONTINUED)

To make the *original* postcondition true could increment left until a[left+1]=x and return left+1 as result (or could decrease middle).

i.e. **r** is the index of the first occurrence of x in a.

We can also strengthen the invariant to reflect that a[right]>x (instead of a[right]≥x). Then, if stop because right-left=1

**What would this tell us about x and a?**



**A new postcondition:**

$0 \le \mathbf{r} \le a.\text{length} \wedge \forall i.\mathbf{int}(0 \le i < \mathbf{r} \rightarrow a[i] < x) \wedge$
$((\mathbf{r} < a.\text{length} \wedge a[\mathbf{r}] = x) \vee \forall i.\mathbf{int}(\mathbf{r} \le i < a.\text{length} \rightarrow a[i] > x))$

(See Problem sheet for details)

## A WARNING

The Binary chop algorithm is presented in many different ways. Possible differences are:

(i) the precondition states that x is known to be in a; this can simplify the postcondition, which can be $0 \le \mathbf{r} < a.\text{length} \wedge a[\mathbf{r}] = x \wedge \forall i{:}\text{int}(0 \le i < \mathbf{r} \rightarrow a[i] < x)$. There is no need to state that elements beyond **r** are ≥x – they must be since a is sorted. For the original postcondition this was required, since a[**r**]≥x could not be used instead of a[**r**]=x, as a[**r**] might not be a valid array access;
(ii) the test between a[middle] and x has three outcomes, depending on whether a[middle]=x, a[middle]<x or a[middle]>x. This allows for the while loop to terminate early in case the value x is encountered, although care must be taken to ensure $\forall i{:}\text{int}(0 \le i < \mathbf{r} \rightarrow a[i] < x)$ is true at the end.
(iii) the right variable indicates the *last* index of the portion of a that has still to be searched. This is in contrast to what was proposed here, where right was the first element *beyond* the part of a still to be searched;
(iv) the left variable indicates the first element in the part of a that has still to be searched. For this variation the initial IF-statement can be dropped. This version is covered in the exercise sheet, and uses a different computation of middle;
(v) sometimes both of (iii) and (iv) are used; the resetting of left or right may then be slightly different, being left=middle+1, or right=middle-1;
(vi) if the postcondition is weakened to $0 \le \mathbf{r} < a.\text{length} \wedge a[\mathbf{r}] = x$ and x is known to be in a (ie just find <u>some</u> index of a), then if the 3-way test is used the while condition can be changed to (right-left>2), and the invariant to $0 \le \text{left} < \text{right}-1 \le a.\text{length}-1$. The initial test should check that the array has at least 2 elements, else it is known the only one must =x.

None of these variations affects very much the efficiency of the algorithm for large arrays a.

## LOOP INVARIANT PROOF STRATEGY (REVISION)

We want to show:
"**if** Pre holds, **and** the code is executed **and** the code terminates, **then** Post will hold".

*We don't discuss termination itself here.*
*We assume we've shown it (by reasoning about the variant)!*

Must show:

Precondition for loop
{Loop Initialisation code}
{Loop code}
{Loop Finalisation code}
Postcondition for loop

To do this, show:

1. *Pre* {Loop Initialisation code} *Invariant*
2. *Invariant & (while condition)* {Loop code} *Invariant*
3. *Invariant & ¬(while condition)* {Loop Finalisation code} *Post*

**A natural deduction view**

**Pre** **Done** ☺
**{Loop Initialisation code}**
**Loop Invariant**

**Loop Invariant (before iteration)**
**(While condition)**
**{Loop statements}**
**Loop Invariant (after iteration)**

**Invariant ∧ not (While condition)** ☺
**{Loop Finalisation code}**
**Post**

**Pre {Init code}{Loop statements}**
**{Final code} Post**

## Specification of Binary Chop (repeated for convenience)

**(assume a=a0 and x=x0 throughout)**

```
int search(int [] a,  int x) {
//
//Pre:        Sorted(a): i.e.,
//            (Pre) ∀i,j:int (0≤i≤j< a.length→a[i]≤a[j])
//Post:       (P1) 0≤ r≤ a.length
//            (P2) ∧ ∀i:int (0≤ i<r → a[i]<x)
//            (P3) ∧ ∀i:int (r ≤ i < a.length → a[i]≥x)
//
//Loop Invariant :   (I1) 0≤left<right≤a.length ∧
//                   (I2)  ∀i:int(0≤i≤left → a[i]<x) ∧
//                   (I3) ∀i:int(right≤i< a.length →a[i]≥x)
//
//Loop Variant:  (Var) right – left-1
```

## THE CODE FOR SEARCH (BINARY CHOP)

```
//(Repeated for convenience)
int search (int [] a, int x) {
// Pre:  Sorted(a)
  int left, mid;
  int right =a.length;
  if ((a.length==0||(a[0]>=x)) return 0; //a[0], …, a[a.length-1]≥x
  left = 0;      //a[left]<x & 0=left<right=a.length
  while  (right-left>1) {
    mid= (left+right) / 2;      // left < mid < right
    if (a[mid]< x)              //Array access is legal
        left = mid;             //a[0], ..., a[left]<x
    else right = mid;           //a[right], ..., a[a.length-1]≥x
    }
  return right;
  }
```

## CONVENTIONS USED (REMINDER)

Let v be a variable used in the code.

v0 is value of v at beginning of method or program (fragment)

*in comments*: v is value of v now

Meaning of "now" depends on our current concern.

| | |
|---|---|
| E.g. | Now = at end of method. |
| | Now = at start of method. |
| | Now = just after an iteration of a loop. |
| | Now = just before first iteration of a loop. |

v1, v2 are names for value of v at some intermediate stage.
*Allows to refer to these values in proofs.*
(You should say what they mean!)

In this problem (binary chop), we'll also assume variables are type **int**

## Informal, but careful, proof that invariant is re-established

Let left1 and right1 be the values of the variables left and right at the start of some iteration and left2, right2 and mid2 the values at the end.

We must show that if at the start of an iteration we have

| | |
|---|---|
| 0≤left1<right1≤a.length | —(2) Inv. (I1) |
| ∀i(0≤i≤left1→a[i]<x) | —(3) Inv. (I2) |
| ∀i(right1≤i<a.length→a[i]≥x) | —(4) Inv. (I3) |
| left1<right1-1 | — while condition is true |

then at the end of the iteration we shall have

(I1') 0≤left2<right2≤a.length

(I2') ∀i(0≤i≤left2→a[i]<x)

(I3') ∀i(right2≤i<a.length→a[i]≥x) — invariant still true

right2-left2-1 < right1-left1-1 — variant decreased

The code ensures left1 < mid2 < right1 — (5) code

We are also given that a is sorted and as no assignments are made it remains sorted. i.e. ∀i,j(0≤i≤j<a.length→a[i]≤a[j]) —(1) pre

## Informal proof (continued) Case 1

**Case 1: a[mid2]<x**

(6a) left2 = mid2 (by code)

(7a) right2 = right1 (by code)

RTS (I1'):

0 ≤ left2 < right2 ≤ a.length <==> 0≤mid2<right1≤a.length

(use 6a and 7a)

True: since 0≤left1 (2) <mid2 (5) <right1(5) ≤a.length (2).

RTS (I2'):

∀i(0≤i≤left2→a[i]<x) <==> ∀i(0≤i≤mid2→a[i]<x)

True : ∀i(0≤i≤mid2→a[i]≤a[mid2]<x) (a is sorted by (1), and case)

RTS (I3'):

∀i(right2≤i<a.length→a[i]≥x) <==> ∀i(right1≤i<a.length→a[i]≥x)

True by (4).

**So the invariant is re-established in this case**

## Informal proof (continued) Case 2

**Case 2: a[mid2]≥x**

(6b) right2 = mid2 (code)

(7b) left2 = left1 (code)

RTS (I1'):

0 ≤ left2 < right2 ≤ a.length <==> 0≤left1<mid2≤a.length.

True: since 0 ≤ left1 (2) <mid2 (5) <right1 (5) ≤ a.length (2).

RTS (I2'):

∀i(0≤i≤left2→a[i]<x) <==> ∀i(0≤i≤left1→a[i]<x) True (3).

RTS (I3'):

∀i(right2≤i<a.length→a[i]≥x) <==> ∀i(mid2≤i<a.length→a[i]≥x).

True: by (1) and the case ∀i(mid2≤i<a.length→a[i]≥a[mid2]≥x)

**So the invariant is re-established.**

We compare our informal proof that the invariant is re-established with a **Natural deduction proof** of the same property.

**Assumptions** (as on slide 45):

| | |
|---|---|
| Sorted(a), i.e., ∀i,j(0≤i≤j<a.length→a[i]≤a[j]) | —(1) pre |
| 0≤left1<right1≤a.length | —(2) Inv. (I1) |
| ∀i(0≤i≤left1→a[i]<x) | —(3) Inv. (I2) |
| ∀i(right1≤i<a.length→a[i]≥x) | —(4) Inv. (I3) |
| left1<mid2<right1 | —(5) code |

(5) implies the while condition is true, so we can now drop that.

We also write down the effect of the loop code in Logic:

(a[mid2]<x ∧ (left2 = mid2) ∧ (right2 = right1))

∨ (a[mid2]≥x ∧ (left2 = left1) ∧ (right2 = mid2)) —(6) code

Use natural deduction to prove the invariant after the iteration:

| | | |
|---|---|---|
| (I1') | 0≤left2 < right2≤a.length | – (a) |
| (I2') | ∀i(0≤i≤left2→a[i]<x) | – (b) |
| (I3') | ∀i(right2≤i<a.length→a[i]≥x) | – (c) |

## THE INFORMAL PROOF IN NATURAL DEDUCTION STYLE

Givens are (1) - (5) and (6):
To show (a), (b) and (c) make a case analysis by (∨E) (using (6)) and then for each case show (a), (b) and (c) and then use (∧I).
Here, **we'll just show (b) for the first case**. The outline structure is:

```
6  (a[mid2]<x∧left2=mid2∧right2 =right1)
               ∨(a[mid2])≥x∧left2 =left1∧right2= mid2)
┌──────────────────────────────────────────────┬─────────────────┐
│7 a[mid2]<x∧left2 = mid2∧ right2 =right1       │a[mid2])≥x       │
│┌────────────────────────────────────────────┐│  ∧left2 =left1  │
││8  sk1 (∀I)  0≤sk1≤left2                     ││  ∧right2= mid2  │
││           <fill in this part ...>          ││                 │
││9             a[sk1]<x                      ││                 │
│└────────────────────────────────────────────┘│                 │
│  10 ∀i(0≤i≤left2→a[i]<x)  (∀→I)   (b)         │(a) ∧ (b) ∧ (c)  │
│  11 (a) ∧ (b) ∧ (c)        (∧ I)              │   (∧ I)         │
└──────────────────────────────────────────────┴─────────────────┘
   12  (a) ∧ (b) ∧ (c)        (∨E, 6)
```

## Natural deduction proof continued

```
7 a[mid2]<x∧left2 = mid2∧ right2 =right1
┌──────────────────────────────────────────────────────────┐
│8  sk1 (∀I)  0≤sk1≤left2                                   │
│13       left2 = mid2  (∧E, 7)                            │
│14       0≤sk1≤mid2  (=sub, 13, 8)                        │
│15       a[sk1]≤a[mid2]   (∀→E, 14, 1 – ie a is sorted)   │
│16       a[mid2]<x  (∧E, 7)                               │
│9        a[sk1]<x   (15,16, transitivity of < and ≤       │
└──────────────────────────────────────────────────────────┘
 10 ∀i(0≤i≤left2→a[i]<x)  (∀→I)   (b)
```

The main difference between this more formal proof and the previous one on slides 46/47 is that here the quantifiers have to be removed in order to show the main implication.

Note also the reason for line 9 - it is still fairly informal (but OK). Other cases can be completed in a similar way (exercise).

## NOTES ON STYLE

The informal proof is similar to the natural deduction proof, but has "obvious" bits omitted (e.g. lines 13 and 16 on slide 50).

Some informality is OK in natural deduction too. Eg:

— Quoting properties of <, ≤, + (e.g., line 9 on slide 50).
  Label with "≤" as justification. Anything reasonable goes.

— Implicitly using types of things
  (e.g., don't bother to keep saying a variable v is **int**).

— x≤y≤z can be used to abbreviate x≤y ∧ y≤z.

— x≤y is equivalent to x<y ∨ x=y. So the following is OK:

```
┌──────────────────────┐
│         x≤y          │
│┌──────────┬─────────┐│
││ x<y      │   x=y   ││
││  ⋮       │    ⋮    ││
││  A       │    A    ││
│└──────────┴─────────┘│
│         A      ∨E    │
└──────────────────────┘
```

## PROS AND CONS OF NATURAL DEDUCTION

+ You (should) know how to do it.
+ You (should) know what to write down.
+ Maximises rigour, and minimises writing.
+ Encourages working backwards
  Line numbers in proof on slide 50 indicate order of steps.

- Can be long and tedious (unless you use "trust me" often — but then it's easy to make mistakes).

**Conclusion:**
You should be able to use natural deduction (if required) *and* do informal proofs (mostly all you'll need).
We'll see examples of both in what follows.

## PROVING THINGS FROM POSTCONDITIONS

A program's specification is expressed as “pre ==> post”.
When we use a program, assuming that pre holds, then what happens should depend solely on post.

**Example**: let's prove from the postcondition of Search that
(∀i(0≤i<a.length→a[i]≥x)) → **r** = 0.

The postcondition (*Post*) was: 0≤**r**≤a.length ∧
∀i:int (0≤i<**r** → a[i]<x) ∧ ∀i:int (**r**≤i<a.length → a[i]≥x).

**(Very) informal proof (a less informal proof follows):**

Assume ∀i(0≤i<a.length→a[i]≥x) ==> all entries in a are ≥ x (a).
*Post* ==> all entries in a before **r** are <x (b).
Hence from (a) and (b) there can be no entries before **r**.
(Any such entry would be both ≥x and < x, a contradiction.)
Since **r**≥0 by *Post* (d), it must be that **r**=0.

## A (slightly) more formal Proof

Show (∀i(0≤i<a.length→a[i]≥x)) → **r** = 0:

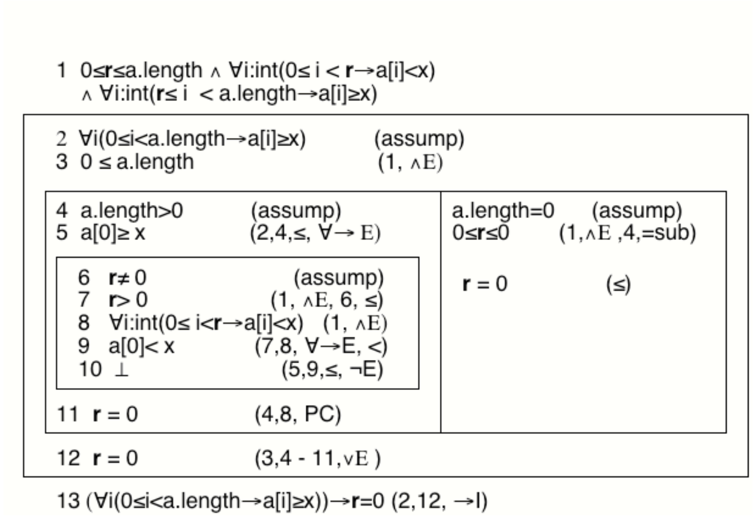
Proof: Assume ∀i(0≤i<a.length→a[i]≥x)
i.e. for every i, a[i]≥x (1).

RTS **r**=0

*Case 1*: a.length>0. By Case a[0] is defined and from (1), a[0]≥x.
Suppose, for contradiction, that **r**≠0 (2);
by *Post* **r** ≥ 0 (3) and ∀i(0≤i<**r**→a[i]<x) (4). From (2), (3) **r**>0
Hence 0≤0<**r** and by (4) a[0]<x, which contradicts a[0]≥x. ∴ **r**=0.
*Case 2:* a.length=0. Hence by *Post* 0≤**r**≤0 ==> **r**=0.

Since a.length≥0, either way **r**=0.
The corresponding formal natural deduction proof is on slide 55.

**Exercise:** prove that the post-condition of Search implies that
(∀i(0≤i<a.length→a[i]<x)) → **r** = a.length.



```
1  0≤r≤a.length ∧ ∀i:int(0≤ i < r→a[i]<x)
   ∧ ∀i:int(r≤ i  < a.length→a[i]≥x)

2  ∀i(0≤i<a.length→a[i]≥x)          (assump)
3  0 ≤ a.length                      (1, ∧E)

4  a.length>0        (assump)          | a.length=0     (assump)
5  a[0]≥ x           (2,4,≤, ∀→ E)     | 0≤r≤0       (1,∧E ,4,=sub)
                                       |
   6   r≠ 0                  (assump)  |  r = 0           (≤)
   7   r> 0            (1, ∧E, 6, ≤)   |
   8   ∀i:int(0≤ i<r→a[i]<x)  (1, ∧E)  |
   9   a[0]< x          (7,8, ∀→E, <)  |
   10  ⊥                  (5,9,≤, ¬E)  |
                                       |
11  r = 0             (4,8, PC)        |

12  r = 0             (3,4 - 11,∨E )

13 (∀i(0≤i<a.length→a[i]≥x))→r=0 (2,12, →I)
```

**Example**: Show ∀x,y:Nats [x≤y → search(a,x)≤ search(a,y)].

As always, try a few test cases. Is it true?

Let a = [1, 3, 6, 12, 20, 23, 23, 25, 32]

| Input x | Input y | Search(a,x) | Search(a,y) | True? |
|---|---|---|---|---|
| 2 | 5 | | | |
| 1 | 2 | | | |
| 21 | 23 | | | |
| 20 | 25 | | | |

**To show:** ∀x,y:Nats [x≤y → search(a,x)≤ search(a,y)].

**Informal proof:** (A formal natural deduction proof is on slide 58.)

Assume x≤y for arbitrary Nats x and y;

let k=search(a,x) and m=search(a,y)

RTS: search(a,x)≤ search(a,y) – i.e. k≤m.

Suppose, for contradiction, that k>m.
The postcondition of search gives lots of properties about m and k:

0≤m≤a.length∧∀i:int(0≤i<m→a[i]<y)∧∀i:int(m≤i<a.length→a[i]≥y)
and
0≤k≤a.length∧∀i:int(0≤i<k→a[i]<x)∧∀i:int(k≤i<a.length→a[i]≥x).

From the "k postcondition", and because 0≤m<k, a[m]<x, and
from the "m postcondition", and because m≤m<k≤a.length, a[m]≥y.
Hence a[m]<x≤y≤a[m], or a[m]<a[m], which is impossible, so k≤m.

**Exercise**: Call the "m properties" (1), (2), (3) and the "k properties" (4), (5), (6). State which properties are used where in the proof.

```
x,y (∀I)   Let k=search(a,x) and m=search(a,y)
:Nat       0  x≤y                                        (Ass)
           1  0≤m≤a.length ∧ ∀i:int(0≤ i < m→a[i]<y)
                 ∧∀i:int(m≤ i  < a.length→a[i]≥y)    (post for m)
           2  0≤k≤a.length ∧ ∀i:int(0≤ i < k→a[i]<x)
                 ∧ ∀i:int(k≤ i  < a.length→a[i]≥x)    (post for k)

           3    k>m                (assump)
           4    k ≤ a.length       (2, ∧E)
           5    m<a.length         (3,4, ≤)
           6    m≤m<a.length       (5, ≤)
           7    a[m]≥y             (1, ∧E, ∀→ E)
           8    m≥0                (1, ∧E)
           9    0≤m<k              (3, 8, ∧I)
           10   a[m]<x             (2, ∧E, ∀→ E)
           11   a[m]<x≤y≤a[m]      (∧I, 10, 7, 0)
           12   ⊥                        (11, <)

           13 k≤m ( PC)

14 ∀x,y:Nat [x≤y → search(a,x) ≤ search(a,y)]  (∀→E)
```

**The exercises suggest other properties to prove about binchop**

## FINAL POINTS

- The usual pitfall with the binary chop algorithm lies in not being quite sure what the values of left and right are supposed to mean.

- Making the specification and the loop invariant precise and being careful about the difference between < and ≤ is the way to avoid this.

- Proofs can be made using just the specification. They can be more or less formal, but still rigorous. If you are unsure, try formalising the steps in natural deduction. But **always**, justify each step you make in a proof.

## Appendix (for fun!) A MISSING FABLE of AESOP

Once upon a time, there was a talking computer, Ms. H, and a rich man, Mr. P. Mr. P trusts Ms. H always to do as he wants. They do business via a kind of contract called a *specification*.

One day, Mr. P asks Ms. H to write him a program to calculate the square root of a given number. She checks that she can assume the number is positive and agrees to do the job.

Their verbal contract is that “provided that the input is positive
the output will be the square root of the input”.
sqrt :: float -> float          --pre: x >= 0
-- post: z*z = x, where z = sqrt x

Note: pre-condition restricts the input - guaranteed by the user;
post-condition constrains the output - guaranteed by programmer.

The story continues ….

Ms. H wants to know from Mr. P what to do if the input is bad. Mr. P is not at all demanding and suggests an error message would be adequate.

```
defensiveSquareroot :: float -> float
-- pre: none
--post (x < 0 & error reported) or
--        (x>=0 & (defensiveSquareroot x)^2 = x)
```

In fact, Mr. P plans to use the square-root program to compute fourth roots and he writes the following code. Mr P also shows that fourthroot satisfies its specification (as long as sqrt satisfies its specification). He also notices that his program and task would be much simpler if sqrt always returned a positive value.

```
fourthroot :: float -> float              -- pre: x >= 0
--post z ^ 4 = x, where z = fourthroot x
fourthroot x
          | y >=0      = sqrt y
          |otherwise  = sqrt (-y)   where y=sqrt x
```

*Does fourthroot work as it should?*
Show forall x: float(if x >=0 then (fourthroot)^4 = x.

Take an arbitrary value for x - call it c.
Then (sqrt c)^2 = c (definition of sqrt).
Put y for (sqrt c), then y^2 = c,  and F for fourthroot c.
Notice that (F^2)^2=F^4.
If y >=0 then F^2 = (sqrt y)^2 = y and (F^2)^2 = y^2 = c.
If y < 0, F^2 = (sqrt (-y))^2 = (-y); (F^2)^2 = (-y)^2 =  y^2 = c.
(Notice (-y) satisfies pre-cond of sqrt.) In both cases F^4 = c.

And then ...

Ms. H tells Mr. P one day that she can only calculate squareroots to within a fixed tolerance - not exactly. Mr. P is very displeased as he has just paid thousands of pounds to have the programs hard-wired into some gadget he is marketing. He storms off, muttering that Ms. H will hear from his lawyers …

But luckily Ms. H had got to the lawyers in advance of the contract and had written this disclaimer:

"This software might do anything at all,
but there again it might not.
Anything Ms. H may say about it is inoperative."

```
sqrt :: float -> float        --pre: x >= 0
--post: |(sqrt x)^2 -x | < tolerance (tolerance still to be negotiated)
```

By changing the contract (in favour of Ms.H) Mr. P must check that *he* can satisfy *his* customer (who might be himself).
He realises that his fourthroots are not exact any more and must check his code once more, using the new post-condition.

Keep in mind the spring model:
Stronger pre-condition/weaker post-condition is good for Ms. H.
(She can assume more and show less.
Mr. P. must check more and gets to use less.)
Weaker pre-condition/stronger post-condition is good for Mr. P.
(He has less to check and gets more.
Ms. H assumes less and must deliver more.)

So far, then, Ms. H and Mr. P  have agreed on a specification with error tolerances ...

Mr. P notices that the program provided always seems to deliver positive squareroots. He remarks on this and it is confirmed by Ms. H. Mr. P is pleased as it means a simpler fourthroot program for him.He prepares for his manufacturing.

Meanwhile, Ms. H (working late into the night) notices that she can get a faster program if she returns the negative square root. (e.g. sqrt = -2 instead of 2.) She goes ahead and changes the program. Since she didn't sign any contracts about the sign of the result, there is no problem ….

But … Mr. P is not happy. He thought he had such a contract …
and now his programs don't work!