Course 141
– Reasoning about Programs –

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.
- Illustrate 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.

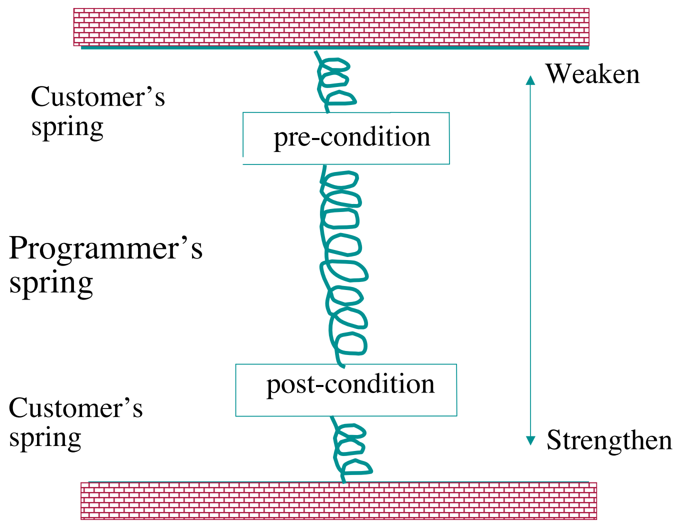
SPECIFICATION – FIRST ATTEMPT

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

In this part of the course, **r** denotes the value returned by the function.

Is that it?
Things to 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)

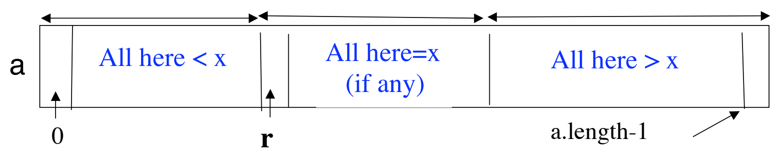
Way 2 is our standard. Then **r** is *the smallest index where the array element is* > x (or a.length if all the elements are < x).

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 and a=a0

// i.e. a=a0 ∧ 0 ≤ **r** ≤ a.length
// ∧ ∀i.**int** (0 ≤ i < **r** → a[i]< x)
// ∧ ∀i.**int** (**r** ≤ i < a.length → a[i] ≥ x)

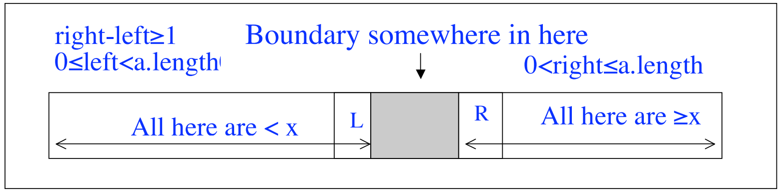
LOOP INVARIANT (REMINDER)

```
//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 must terminate
if v1 > v2 > v3 >.... > vk > ... > I in the loop.
(vi=value of v inside loop for ith time;
I is a fixed int, usally 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*.

(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?**

This page is deliberately left blank

Proof of Property (i). (*Informal*)
Given 0≤left<a.length (1), 0<right≤a.length (2), ∀i(0≤i≤left→a[i]<x) (3) and ∀i(right≤i<a.length→a[i]≥x) (4), we are required to show (RTS) that left<right.
We can use proof by contradiction: Suppose left≥right.
<u>Case 1</u>: If right=a.length then by assumption left≥a.length, which contradicts (1).
<u>Case 2</u>: If right<a.length, then by assumption, (1) and (2) 0<right≤left<a.length.
By (4) a[left]≥x and by (3) a[left]<x, contradiction.

Proof of Property (ii) (*Informal*) Given 0<left+1=right≤a.length, x is in a, and (3) and (4).
RTS (a) right<a.length and (b) a[right]=x.
(a) Suppose for contradiction that right=a.length. Hence left=a.length-1 and by (3) all elements in a are <x which contradicts x is in a.
(b) Assume result (a) and suppose for contradiction that a[right]≠x. From (4) a[right]≥x, so a[right]>x and since right=left+1, a[left+1]>x. (5). By (3) we're also given a[left]<x (6).
By sortedness, ∀i(0≤i≤left→a[i]≤a[left]) and ∀i(a.length>i≥left+1→a[i]≥a[left+1]).
Hence by (5) and (6) ∀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 7. For (ii) the property that 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. Perhaps you see why a tool for doing all this would be useful!

TOWARDS THE CODE

Loop invariant:

$0 \leq left < right \leq a.length \wedge a = a0$
$\wedge\ \forall i.\mathbf{int}\ (0 \leq i \leq left \rightarrow a[i] < x)$
$\wedge\ \forall i.\mathbf{int}\ (right \leq i < a.length \rightarrow a[i] \geq x)$

Loop variant: right-left-1
(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 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? – at least 1 error; consider some special cases.

‡ Second try:

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

Proof invariant is initially established (for non-empty a and a[0]<x)

1st conjunct: RTS (Required To Show)
$0 \leq 0 < a.length \leq a.length$ (ie substitute for values of left etc.)
True by arithmetic and assumption that here a is non-empty

2nd conjunct: if i=0 then a[i] = a[0]<x and $0 \leq i \leq 0$ so implication true.
For all other i condition of implication is false.

3rd conjunct: There is no i 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.

Given: After the else condition of the if-statement, we know
left=0, a.length>0, right=a.length and a[0]=a[left]<x (note a[0] is defined).

1st conjunct: RTS $0 \leq left < right \leq a.length \iff 0 \leq 0 < a.length \leq a.length$.
$0 \leq 0$ and $a.length \leq a.length$ by arithmetic. $0 < a.length$ is given.

2nd conjunct: RTS $\forall i(0 \leq i \leq left \rightarrow a[i] < x)$.
Let I be an arbitrary int s.t. (such that) $0 \leq I \leq left$ (1).
Then RTS a[I]<x. There are two cases: either I=0 or $0 < I \leq left$ (by (1)).
Case 1: I=0. a[I]=a[0]<x (given).
Case 2. $0 < I \leq left \iff 0 < I \leq 0 \iff \bot \implies a[I] < x$ ($\bot$E).
Either way a[I]<x is shown as required.

3rd conjunct: RTS $\forall i(right \leq i < a.length \rightarrow a[i] \geq x)$.
Let I be an arbitrary int s.t. $right \leq I < a.length \iff$
$a.length \leq I < a.length \iff \bot \implies a[I] \geq x$ ($\bot$E).

Question: In the 2nd and 3rd Conjuncts what ND rule was used at the outer level?
Answer: The $\forall \rightarrow$I rule.

Loop Finalization (establishing Post)

//Post: (1) $0 \le \mathbf{r} \le a.length$
// (2) $\wedge\ \forall i.\mathbf{int}\ (0 \le i < \mathbf{r} \rightarrow a[i] < x)$
// (3) $\wedge\ \forall i.\mathbf{int}\ (\mathbf{r} \le i < a.length \rightarrow a[i] \ge x)$

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)

(1) <==> 0≤0≤a.length <==> true (arithmetic and Givens)

(2)<==>$\forall i.\mathbf{int}(0 \le i < 0 \rightarrow a[i] < x)$<==>true as for all i, 0≤i<0 is false

(3) <==> $\forall i.\mathbf{int}\ (0 \le i < a.length \rightarrow a[i] \ge x)$. There are 2 cases:

(i) *a.length=0* ==> 0≤i<0 is false for every i==>implication true

(ii) *a.length>0*: a[0]≥x and a is sorted ==>x≤a[0]≤a[i] for every i.

Case 2 (exit after loop) right-left≤1; **r**=right.

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

right-left≤1 ==> right≤left+1. *Inv.* ==> right≥left+1; ∴ right=left+1.

Substitute **r** for right and **r-1** for left in the invariant – gives Post.

eg in $\forall i.\mathbf{int}\ (0 \le i \le left \rightarrow a[i] < x)$ put **r-1** for left, then

$\forall i.\mathbf{int}\ (0 \le i \le \mathbf{r-1} \rightarrow a[i] < x)$ ==> $\forall i.\mathbf{int}\ (0 \le i < \mathbf{r} \rightarrow a[i] < x)$ (2)

RE-ESTABLISHING THE INVARIANT –

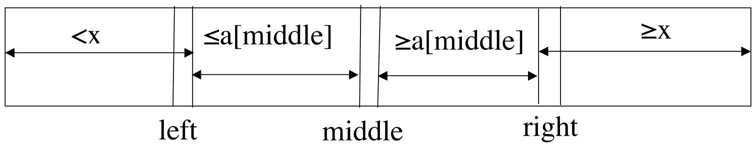
The idea is to define 'middle' as (left+right) / 2.

If a[middle] < x, then we can replace left by middle.
And if i ≤ left then 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)



if a[middle]≥x then right=middle

if a[middle<x then left = middle

RE-ESTABLISHING THE INVARIANT – CTD.

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.

c) that the variant, right–left-1, has strictly decreased.

Given the invariant, (0≤left<right≤a.length), they all follow simply from the following fact, which could usefully be included as a comment:

if left ≤ right-2, then left < middle < right

Exercise: Show formally that the Loop Finalization implies Post and 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)

Case 1: (exit before loop starts) **r**=0=a.length. RTS (1), (2) and (3) of Slide 17.
(1)<==> 0≤**r**≤a.length<==>0≤0≤0 <==>True by arithmetic
(2)<==> ∀i(0≤i<**r**→ a[i]<x)<==>∀i(0≤i<0→ a[i]<x)<==>True since for all i, 0≤i<0 is False.
(3)<==> ∀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. RTS (1), (2) and (3) of Slide 17.
(1)<==> 0≤**r**≤a.length<==>0≤0≤a.length <==>True by arithmetic and the case assumptions
(2)<==> ∀i(0≤i<**r**→ a[i]<x)<==>∀i(0≤i<0→ a[i]<x)<==>True since for all i, 0≤i<0 is False.
(3)<==> ∀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) right-left≤1 and **r**=right. RTS (1), (2) and (3) of Slide 17.

Given Invariant:
(I1) 0≤left<right≤a.length; (I2) ∀i(0≤i≤left→ a[i]<x) (I3) ∀i(right≤i<a.length→a[i]≥x)

Note: right-left≤1<==>right≤left+1 and left<right (by I1) <==> left+1≤right; ∴ left+1=right

(1) <==> 0≤**r**≤a.length<==>0≤right≤a.length <==> 0≤left+1≤a.length.
By (I1) 0≤left ==> 0≤left+1 and left<a.length<==>left+1≤a.length.
(2)<==> ∀i(0≤i<**r**→ a[i]<x)<==>∀i(0≤i<right→ a[i]<x)<==> ∀i(0≤i<left-1→ a[i]<x)
<==>∀i(0≤i≤left→ a[i]<x)<==>True by (I2).
(3)<==> ∀i(**r**≤i<a.length→ a[i]≥x)<==>∀i(right≤i<a.length→ a[i]≥x)<==>True by (I3).

We can show the Properties a), b), c) as follows.
First we show left≤right-2 → left<middle<right using middle = (left+right)/2;
Suppose left≤right-2. RTS left<middle and middle<right.
left+2≤right ==> middle≥(left+left+2)/2=left+1 <==> middle>left, and
middle≤(right-2+right)/2=right-1<==> middle <right.

Next we show properties a), b), c)

Given: (G1) middle = (left+right)/2; (G2) left≤right-2 → left<middle<right;
(G3) 0≤left<right≤a.length (from Invariant (I1)) and (G4) left≤right-2 (loop test);
(G2)+(G4)==> left<middle<right (G5).

a) <==> 0≤middle<a.length. This follows from (G3) and (G5).

Let left1 and right1 be the values of left/right before the reassignment of left or right and left/right the values after.
Either (i) left=middle and right=right1, or (ii) left=left1 and right=middle.

b) (G6) 0≤left1<middle<right1≤a.length follows from (G3) and (G5).
(i): RTS 0≤left<right≤a.length<==>0≤middle<right1≤a.length.
Follows from a) and (G6).
(ii) is similar.
Hence in both cases right-left-1≥0.

c) Variant before loop code = right1-left1-1. Variant after loop code = right-left-1.
(i) right-left-1=right1-middle-1<right1-left1-1.
(ii) right-left-1=middle-left1-1<right1-left1-1.
So in both cases variant≥0,from (b), and variant decreases, from (c).

HOW DO WE KNOW THAT LEFT < MIDDLE < RIGHT?

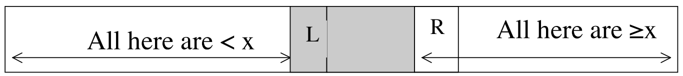
Remember we are assuming left ≤ right-2. Then

middle = (left+right) / 2 ≥ (left+left+2) / 2 = left+1

middle = (left+right) / 2 ≤ (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 can 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 exercises guide you to fill in the details.

For yet another version, 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 13
    // 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;
}
```


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.

VARIATION 1: CHANGE OF PRE/POST

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

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

We can then strengthen the postcondition, which was:

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

to

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

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?

VARIATION 1 (CONTINUED)

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

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

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

Old-Post: $0 \le \mathbf{r} \le a.length \wedge$

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

to new-Post:(4) $0 \le \mathbf{r} < a.length \wedge a[\mathbf{r}]=x \wedge \forall i.\mathbf{int}(0 \le i < \mathbf{r} \rightarrow a[i] < x)$

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

From (3) $a[\mathbf{r}] \ge x$ and $\forall i.\mathbf{int}(0 \le i < \mathbf{r} \rightarrow a[i] < x)$

If $a[\mathbf{r}] > x$, then by (1) $\forall i.\mathbf{int}(\mathbf{r} \le i < a.length \rightarrow a[i] \ge a[\mathbf{r}] > x)$

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

This contradicts (2). Therefore $a[\mathbf{r}]=x$

Case 2: $r=a.length$. Then (3) contradicts (2) 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

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

Q: Why is this not now guaranteed always to be true?

HINT: Consider an array in which all values =x.

All elements are equal to x

VARIATION 2 (CONTINUED)

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

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 also know that a[right]≠x.

What would this tell us about x and a?

A new postcondition:

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

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 $0 \le \mathbf{r} < a.length \wedge a[\mathbf{r}]=x \wedge \forall i{:}int(0 \le i < \mathbf{r} \rightarrow a[i] < x)$. 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 $a[\mathbf{r}] \ge x$ could not be used instead of $a[\mathbf{r}]=x$, as $a[\mathbf{r}]$ 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 $\forall i{:}int(0 \le i < \mathbf{r} \rightarrow a[i] < x$ 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 $0 \le \mathbf{r} < a.length \wedge a[\mathbf{r}]=x$ 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 $0 \le left < right-1 \le a.length-1$. 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.

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 $\leq$ is the way to avoid this.