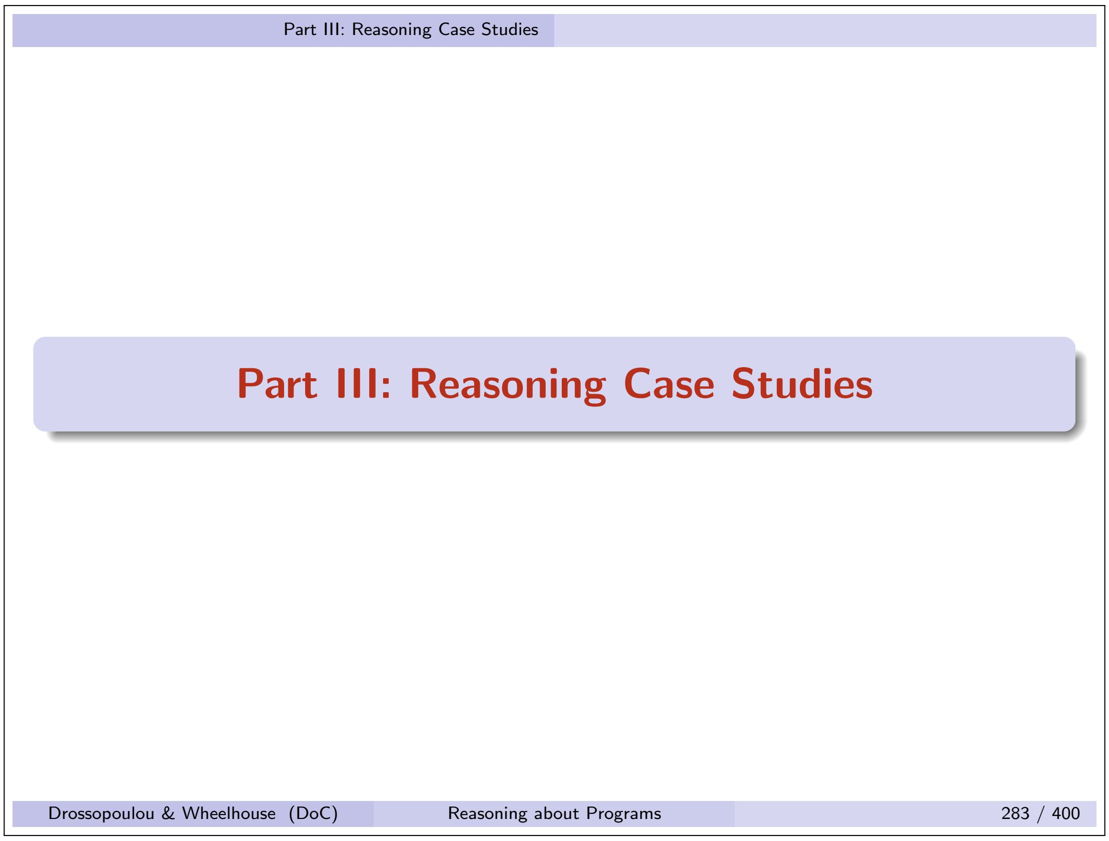
3 Part III: Reasoning Case Studies

Slide 283



Part III: Reasoning Case Studies

Slide 284

Part III – Outline

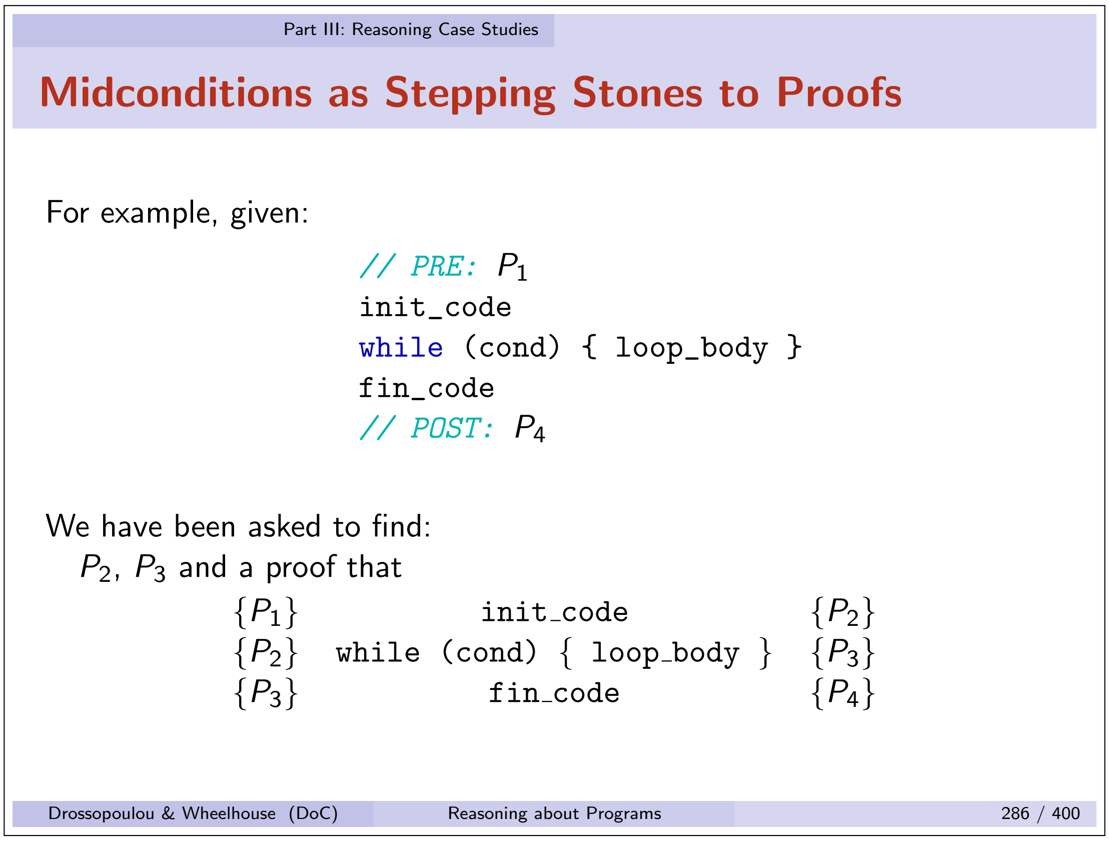
Slide 285

Using Reasoning to Construct Algorithms

Using midconditions and invaraints as stepping stones to *prove* code

v.s.

Using midconditions and invaraints as stepping stones to *design* code



Midconditions as Stepping Stones to Proofs

For example, given:

```
// PRE: P_1
init_code
while (cond) { loop_body }
fin_code
// POST: P_4
```

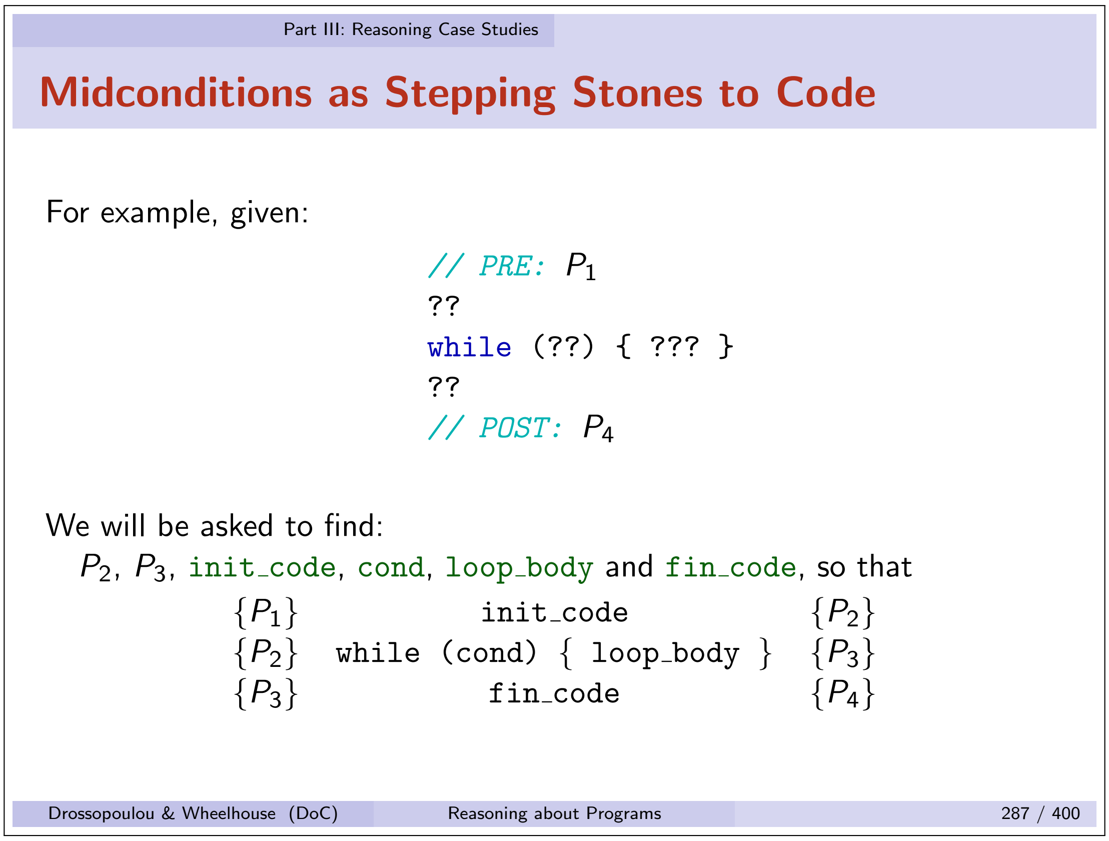
We have been asked to find:

P_2, P_3 and a proof that

$$\begin{array}{ccc} \{P_1\} & \texttt{init\_code} & \{P_2\} \\ \{P_2\} & \texttt{while (cond) \{ loop\_body \}} & \{P_3\} \\ \{P_3\} & \texttt{fin\_code} & \{P_4\} \end{array}$$

So far in this course, we have been given some code and a specification for this code and we have been asked to find appropriate mid-conditions and invariants that alllow us to prove that the code satisfies this specification.

Slide 287



Midconditions as Stepping Stones to Code

For example, given:

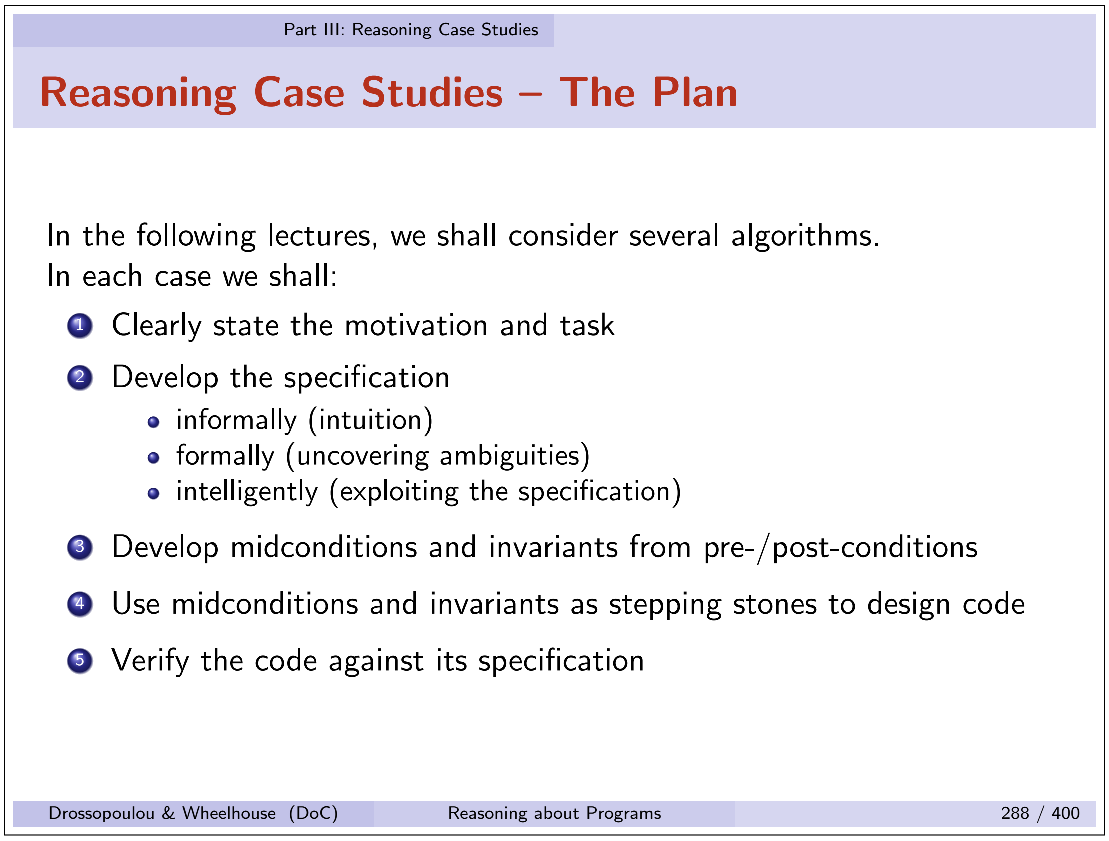
```
// PRE: P1
??
while (??) { ??? }
??
// POST: P4
```

We will be asked to find:
P_2, P_3, `init_code`, `cond`, `loop_body` and `fin_code`, so that

$\{P_1\}$	`init_code`	$\{P_2\}$
$\{P_2\}$	`while (cond) { loop_body }`	$\{P_3\}$
$\{P_3\}$	`fin_code`	$\{P_4\}$

Now, we will be given a specification and be asked to find appropriate mid-conditions and invariants which we can then use to construct the code.

Slide 288



Reasoning Case Studies – The Plan

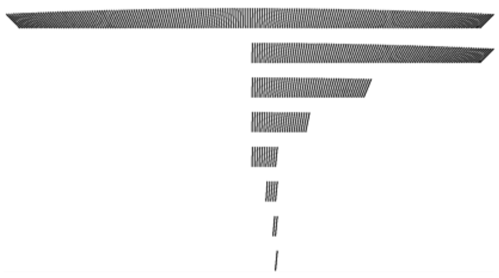
In the following lectures, we shall consider several algorithms.
In each case we shall:

1. Clearly state the motivation and task
2. Develop the specification
   - informally (intuition)
   - formally (uncovering ambiguities)
   - intelligently (exploiting the specification)
3. Develop midconditions and invariants from pre-/post-conditions
4. Use midconditions and invariants as stepping stones to design code
5. Verify the code against its specification

3.1 Binary Search

Slide 289

Binary Search



Slide 290

Binary Search - Motivation

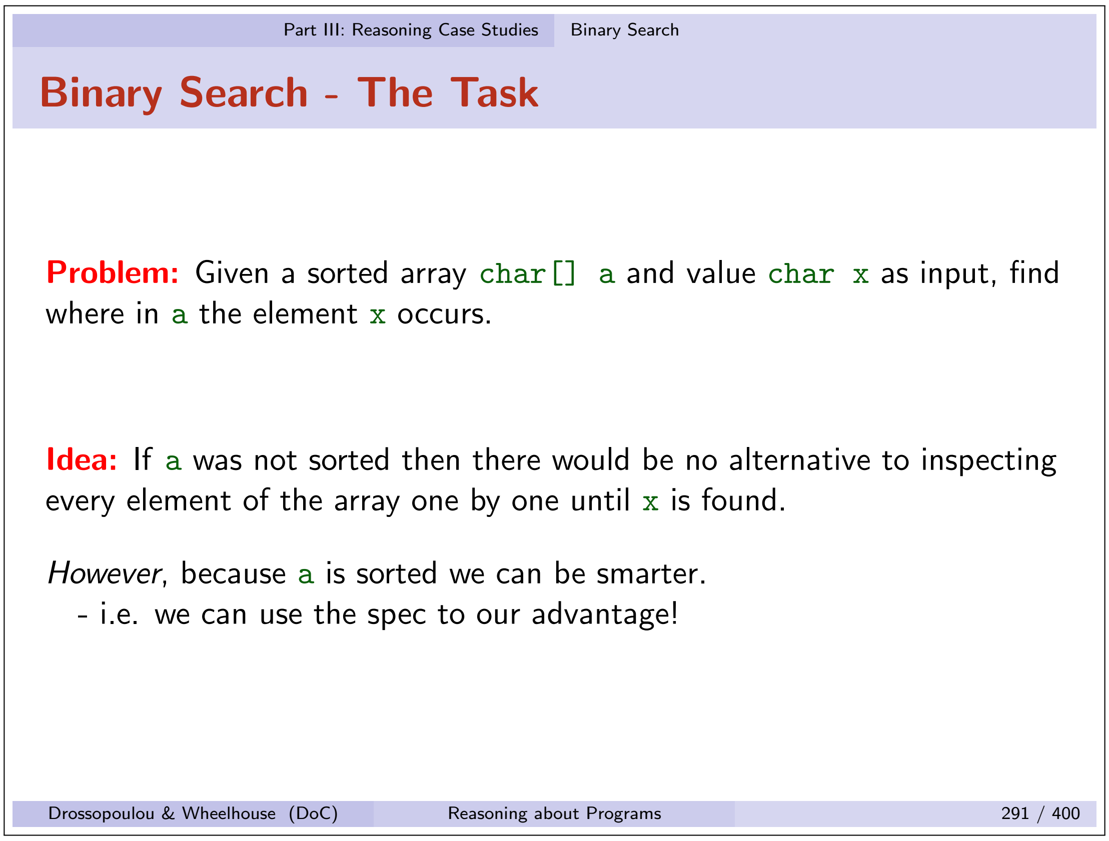
Although the basic idea of binary search is comparatively straightforward, the details can be surprisingly tricky

- Donald Knuth

When Jon Bentley assigned it as a problem in a course for professional programmers, he found that an astounding ninety percent failed to code a binary search correctly after several hours of working on it, and another study shows that accurate code for it is only found in five out of twenty textbooks. Furthermore, Bentley's own implementation of binary search, published in his 1986 book Programming Pearls, contains an error that remained undetected for over twenty years.

- `http://en.wikipedia.org/wiki/Binary_search_algorithm`

Slide 291



Binary Search - The Task

Problem: Given a sorted array `char[] a` and value `char x` as input, find where in `a` the element `x` occurs.

Idea: If `a` was not sorted then there would be no alternative to inspecting every element of the array one by one until `x` is found.

However, because `a` is sorted we can be smarter.
- i.e. we can use the spec to our advantage!

Note: The binary search task can actually be defined for an array of any type that has a less-then-or-equal relation $\leq$.

Slide 292

Binary Search - Pseudo Code

If we assume that a is sorted in ascending (alphabetical) order, then we can use the following rough algorithm:

1. Set the search range to be from 0 to `a.length`
2. Look at the element of a half way along the search range.

3a. If it is x, then we have found it and are done.

3b. If it is bigger than x, then x must be in the first half.
Set the upper bound of the search range to the mid-point.

3c. If it is smaller than x, then x must be in the second half.
Set the lower bound of the search range to the mid-point.

4. Repeat this process from step 2 until x is found.

In case 3a the search terminates and in cases 3b and 3c we cut the search area by a factor of 2.

Slide 293

Binary Search - why don't you have a go?

It probably looks something like...

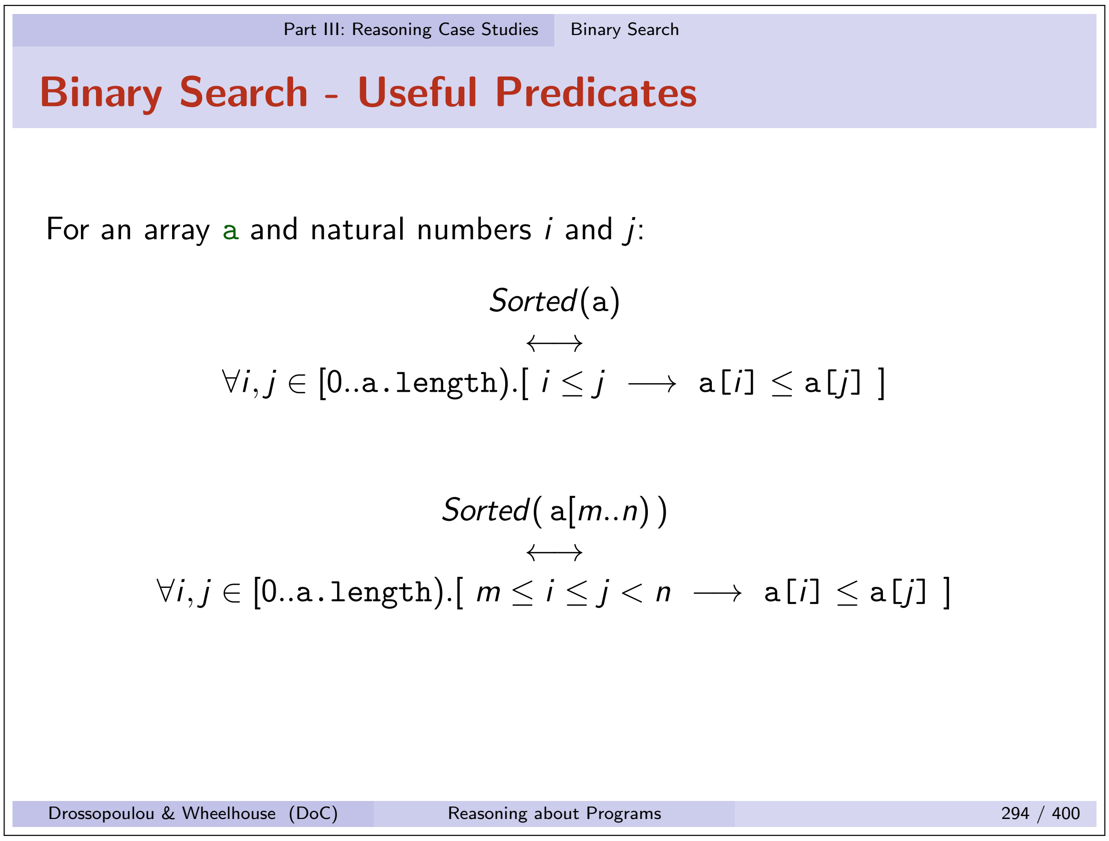
```
int search(char[] a, char x){
  int left = ...   // 1, 0 or -1
  int right = ... // a.length or a.length -1
  while (...) {   // left != right or left < right or ...
    int mid = (left + right) / 2
    if (...) {    // a[mid] <= x or x > a[mid] or ...
      left = mid;
    } else {
      right = mid;
    }
  }
  return ...      // left or right or ...
}
```

Why not try to write both a recursive and an iterative version of the algorithm? Later in this course, you should be able to come back to your code and see if it can be verified.

If we don't consider the method's specification or invariants, then there is actually quite a lot of choice in how we track progress and test for termination and case splits.

We shall shortly see that taking a more principled approach, developing the method's body *systematically* from its specification, leads to less variance in the final code.

Slide 294



Binary Search - Useful Predicates

For an array a and natural numbers i and j:

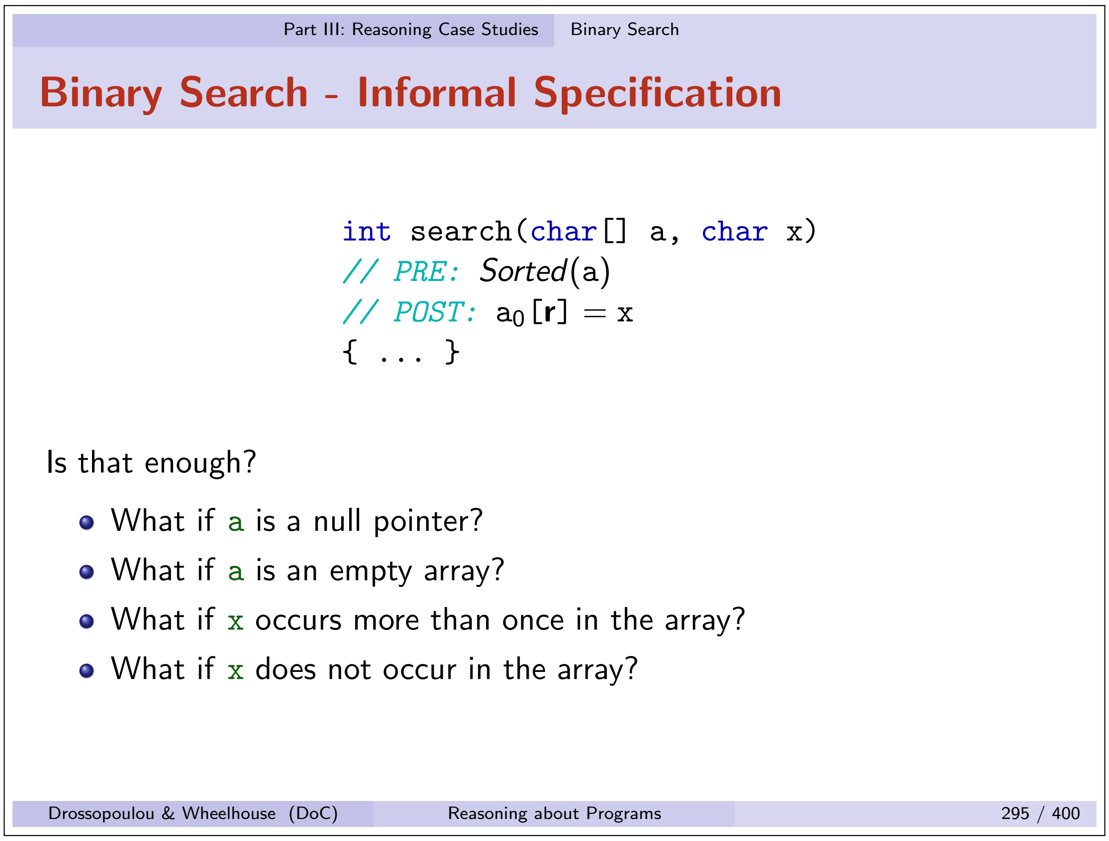
$$Sorted(\texttt{a}) \longleftrightarrow \forall i,j \in [0..\texttt{a.length}).[\ i \le j \longrightarrow \texttt{a}[i] \le \texttt{a}[j]\ ]$$

$$Sorted(\ \texttt{a}[m..n)\ ) \longleftrightarrow \forall i,j \in [0..\texttt{a.length}).[\ m \le i \le j < n \longrightarrow \texttt{a}[i] \le \texttt{a}[j]\ ]$$

The Binary Search algorithm only works on a sorted array. To make our reasoning a little easier we define a predicate that describes when an array is sorted.

The second form of the *Sorted* predicate utilises our array slice notation to allow us to describe that sub-parts of an array are sorted.

Slide 295



Binary Search - Informal Specification

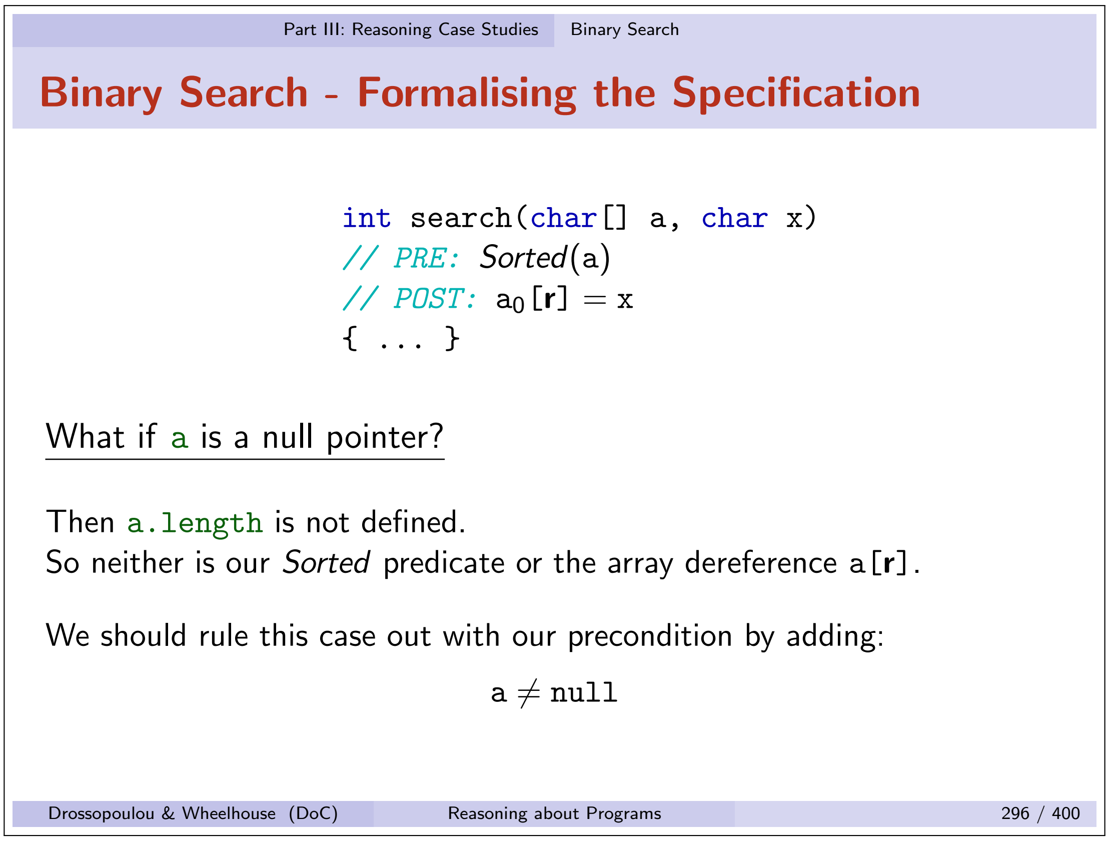
```
int search(char[] a, char x)
// PRE: Sorted(a)
// POST: a₀[r] = x
{ ... }
```

Is that enough?

- What if a is a null pointer?
- What if a is an empty array?
- What if x occurs more than once in the array?
- What if x does not occur in the array?

Remember that **r** stands for the returned result of the method (in this case an `int`).

Also note that we can use `x` in the postcondition and not $\mathtt{x}_0$ as Java is call by value for character method arguments, meaning it is not possible to overwrite the passed in variable `x`.



Binary Search - Formalising the Specification

```
int search(char[] a, char x)
// PRE: Sorted(a)
// POST: a₀[r] = x
{ ... }
```

What if a is a null pointer?

Then a.length is not defined.
So neither is our *Sorted* predicate or the array dereference a[r].

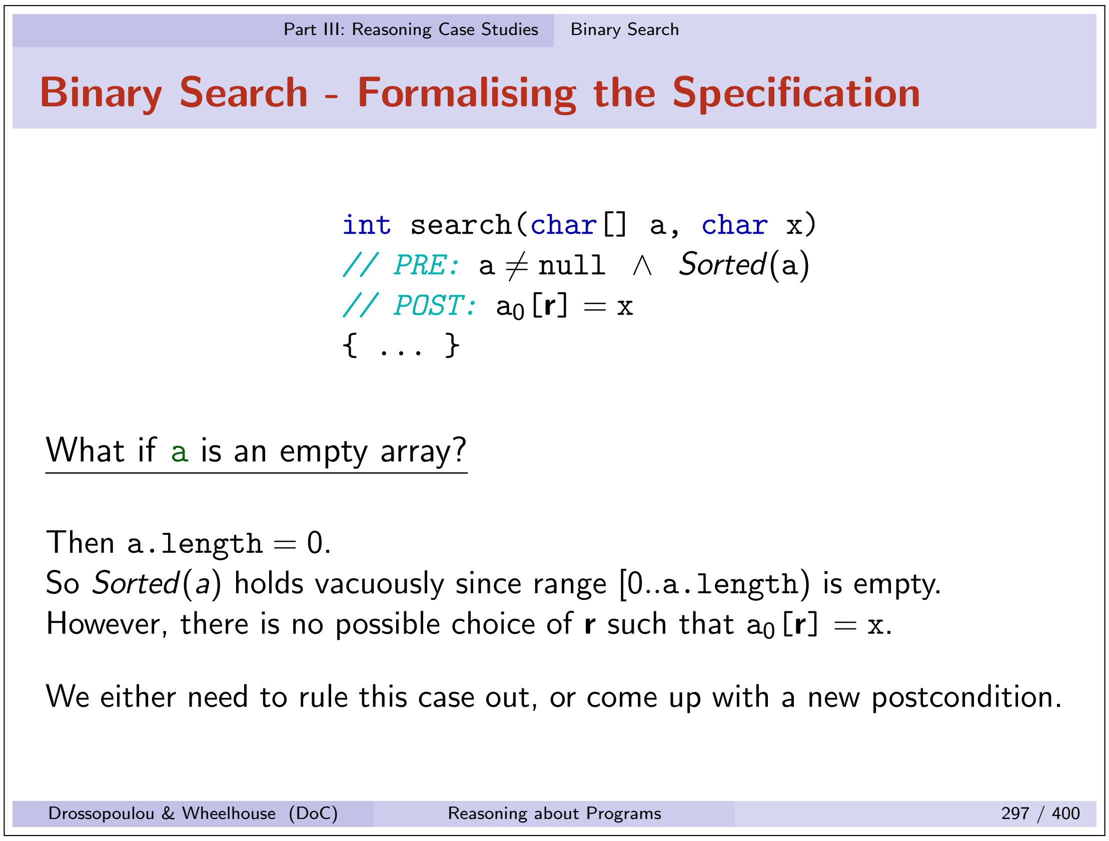
We should rule this case out with our precondition by adding:

$$a \neq \texttt{null}$$

An alternative to explicitly ruling out this case in our precondition would be to adapt our specification to be null pointer tolerant. For example, we could write:

```
int search(char[] a, char x)
// PRE: $a \neq \texttt{null} \longrightarrow Sorted(a)$
// POST: $a_0 \neq \texttt{null} \longrightarrow a_0[r] = x$
//       $\land\ a_0 = \texttt{null} \longrightarrow r = -1$
{ ... }
```

Slide 297



Binary Search - Formalising the Specification

```
int search(char[] a, char x)
// PRE: a ≠ null  ∧  Sorted(a)
// POST: a₀[r] = x
{ ... }
```

What if a is an empty array?

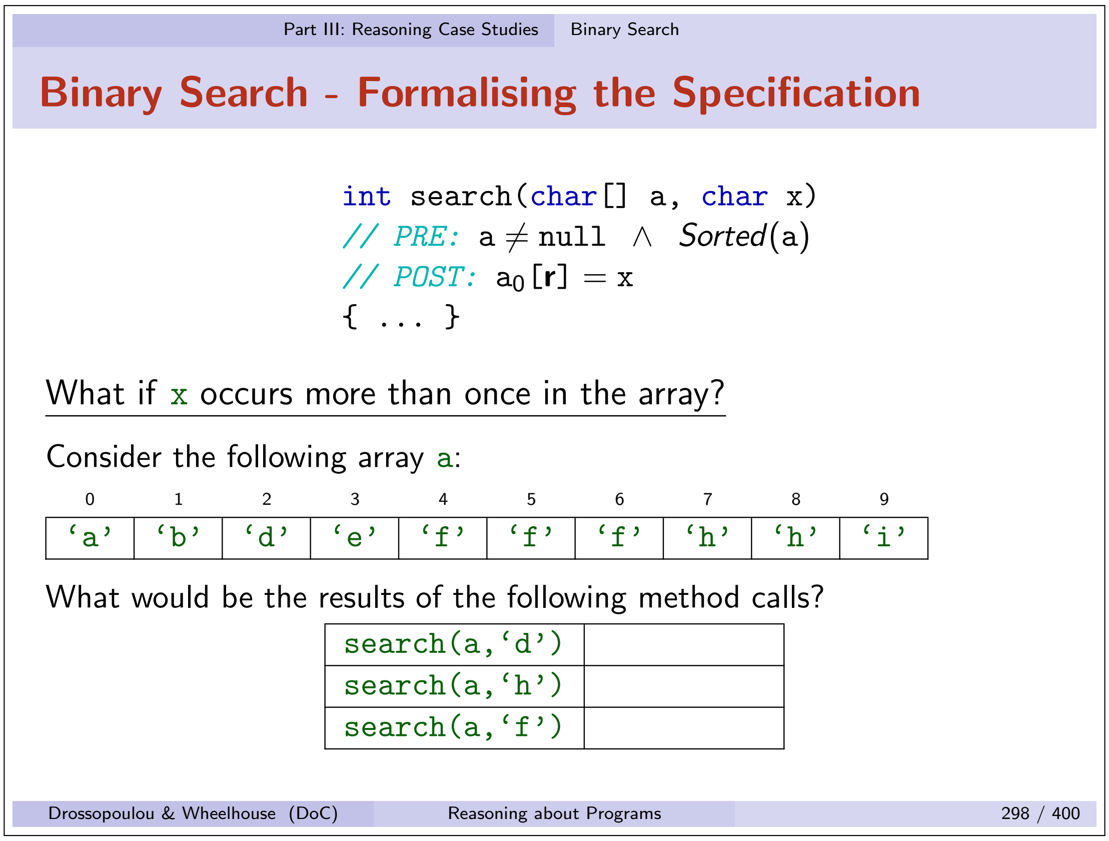
Then $\texttt{a.length} = 0$.
So $\mathit{Sorted}(a)$ holds vacuously since range $[0..\texttt{a.length})$ is empty.
However, there is no possible choice of $\mathbf{r}$ such that $\texttt{a}_0[\mathbf{r}] = \texttt{x}$.

We either need to rule this case out, or come up with a new postcondition.

We could rule this case out with our precondition, by adding:

$$\texttt{a.length} > 0$$

However, notice that $\texttt{a} \neq \texttt{null}$ itself does not rule out the case of the empty array $\texttt{a} = \texttt{[]}$.



Binary Search - Formalising the Specification

```
int search(char[] a, char x)
// PRE: a ≠ null ∧ Sorted(a)
// POST: a₀[r] = x
{ ... }
```

What if `x` occurs more than once in the array?

Consider the following array `a`:

0	1	2	3	4	5	6	7	8	9
'a'	'b'	'd'	'e'	'f'	'f'	'f'	'h'	'h'	'i'

What would be the results of the following method calls?

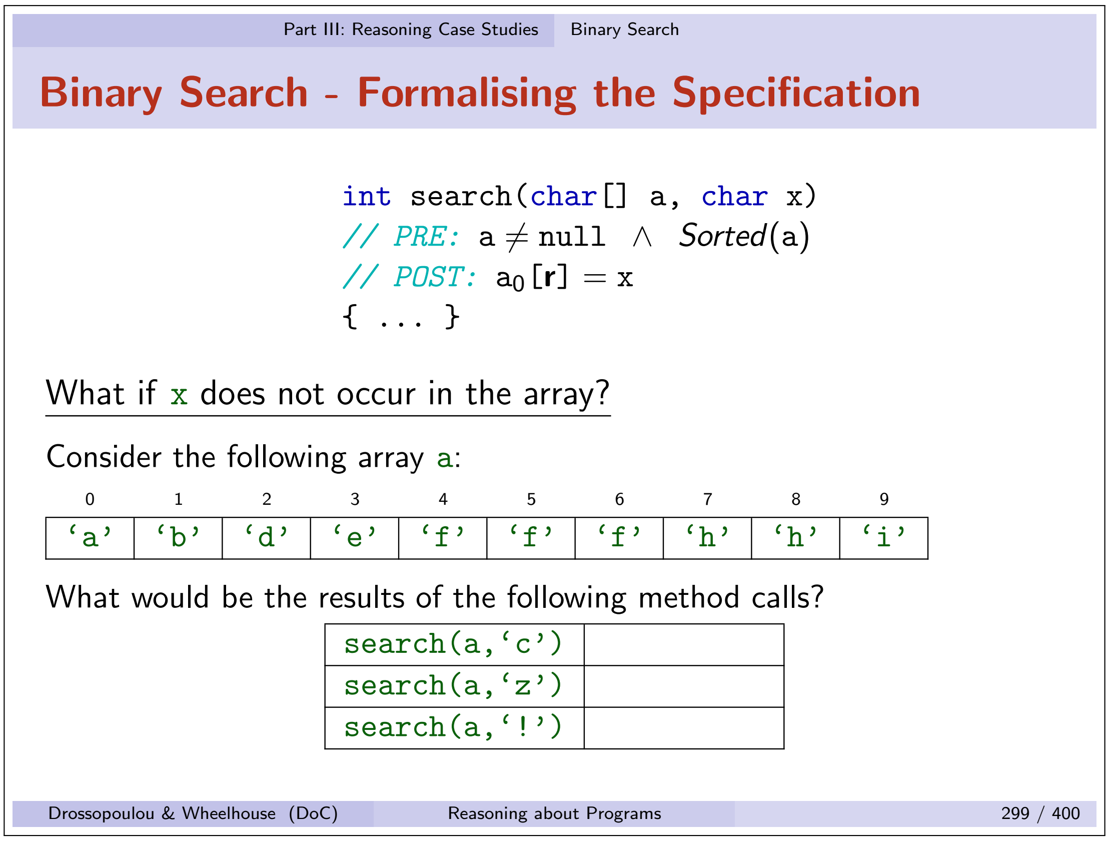
`search(a,'d')`	
`search(a,'h')`	
`search(a,'f')`	

When `x` appears just once in the array the result is unequivocal. However, if `x` appears more than once in the array, then we could reasonably return any index of an occurence.

`search(a,'d')`	2
`search(a,'h')`	7 or 8
`search(a,'f')`	4, 5 or 6

It would probably be more helpful to the method caller if we returned the index of either the first or last occurence of `x`.

Slide 299



Binary Search - Formalising the Specification

```
int search(char[] a, char x)
// PRE: a ≠ null  ∧  Sorted(a)
// POST: a₀[r] = x
{ ... }
```

What if `x` does not occur in the array?

Consider the following array `a`:

0	1	2	3	4	5	6	7	8	9
‘a’	‘b’	‘d’	‘e’	‘f’	‘f’	‘f’	‘h’	‘h’	‘i’

What would be the results of the following method calls?

`search(a,‘c’)`	
`search(a,‘z’)`	
`search(a,‘!’)`	

If `x` does not appear in the array, then we could either return an out of range index (such as `-1` or `a.length`), or the index of where the item ought to fit into the array (such as just before or just after **r**).

`search(a,‘c’)`	-1, 1, 2 or 10
`search(a,‘z’)`	-1, 9 or 10
`search(a,‘!’)`	-1, 0 or 10

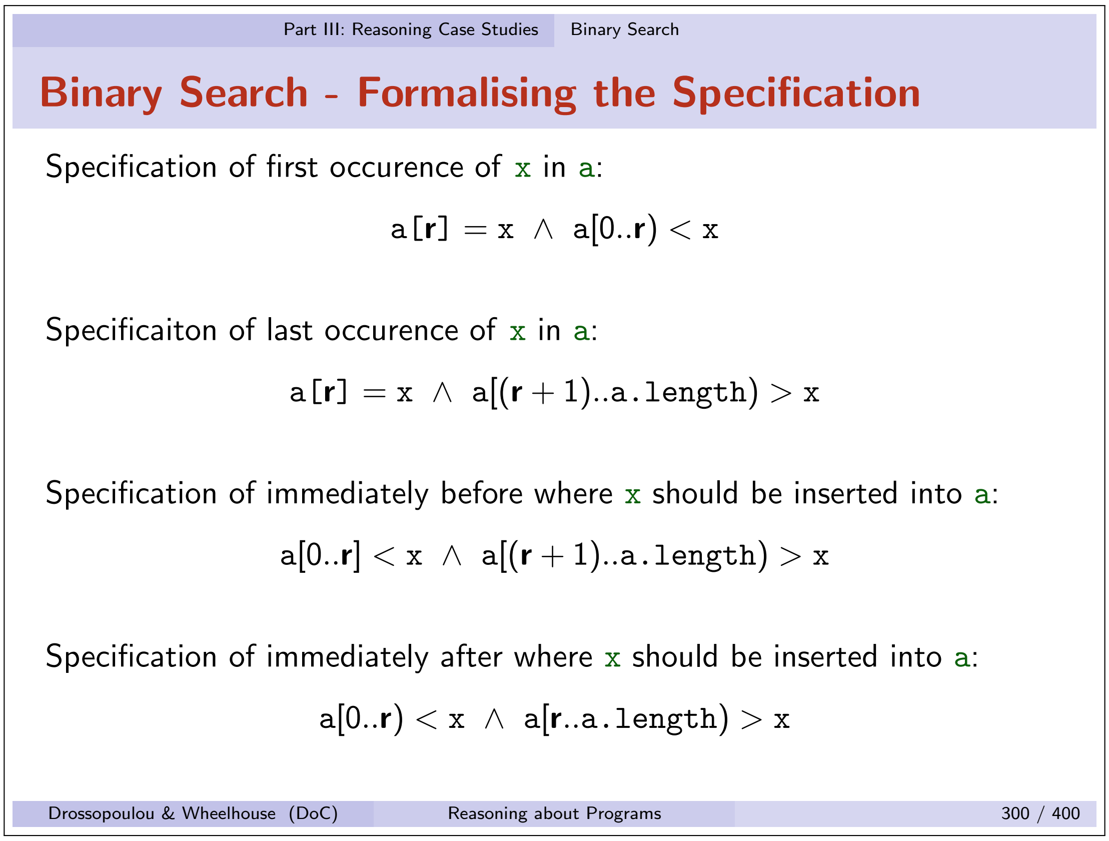
Whichever option we choose, the method caller is going to have to do some post-processing to determine what the return value of the method call means and if it will be safe to dereference.

A test such as:

```
  0 <= r && r < a.length && a[r] == x
```

will determine if the search actually found `x` in the array with either option.

Note that the third conjunct is not necessary with the first option and that we are relying on the lazy evaluation of the expression to avoid dereferencing `a` at an invalid index.



Binary Search - Formalising the Specification

Specification of first occurence of x in a:

$$\texttt{a}[\mathbf{r}] = \texttt{x} \ \wedge \ \texttt{a}[0..\mathbf{r}) < \texttt{x}$$

Specificaiton of last occurence of x in a:

$$\texttt{a}[\mathbf{r}] = \texttt{x} \ \wedge \ \texttt{a}[(\mathbf{r}+1)..\texttt{a.length}) > \texttt{x}$$

Specification of immediately before where x should be inserted into a:

$$\texttt{a}[0..\mathbf{r}] < \texttt{x} \ \wedge \ \texttt{a}[(\mathbf{r}+1)..\texttt{a.length}) > \texttt{x}$$

Specification of immediately after where x should be inserted into a:

$$\texttt{a}[0..\mathbf{r}) < \texttt{x} \ \wedge \ \texttt{a}[\mathbf{r}..\texttt{a.length}) > \texttt{x}$$

We use the fact that the array is sorted and combine the first and last options, using our array slice notation, to get:

$$\texttt{a}[0..\mathbf{r}) < \texttt{x} \leq \texttt{a}[\mathbf{r}..\texttt{a.length})$$

We shall see that this provides the most information about the state of the array after the method call.

Interesting Question:

Does $\texttt{a}[0..\mathbf{r}) < \texttt{x} \leq \texttt{a}[\mathbf{r}..\texttt{a.length})$ imply that $\mathbf{r} \in [0..\texttt{a.length})$?

Answer:

No.

Recall the definition of our array slice notation:

$$\texttt{a}[m..n) \leq x \longleftrightarrow \forall i \in \mathbb{N}[\ m \leq i < n \ \wedge \ 0 \leq i < \texttt{a.length} \ \longrightarrow \ \texttt{a}[i] \leq x\ ]$$

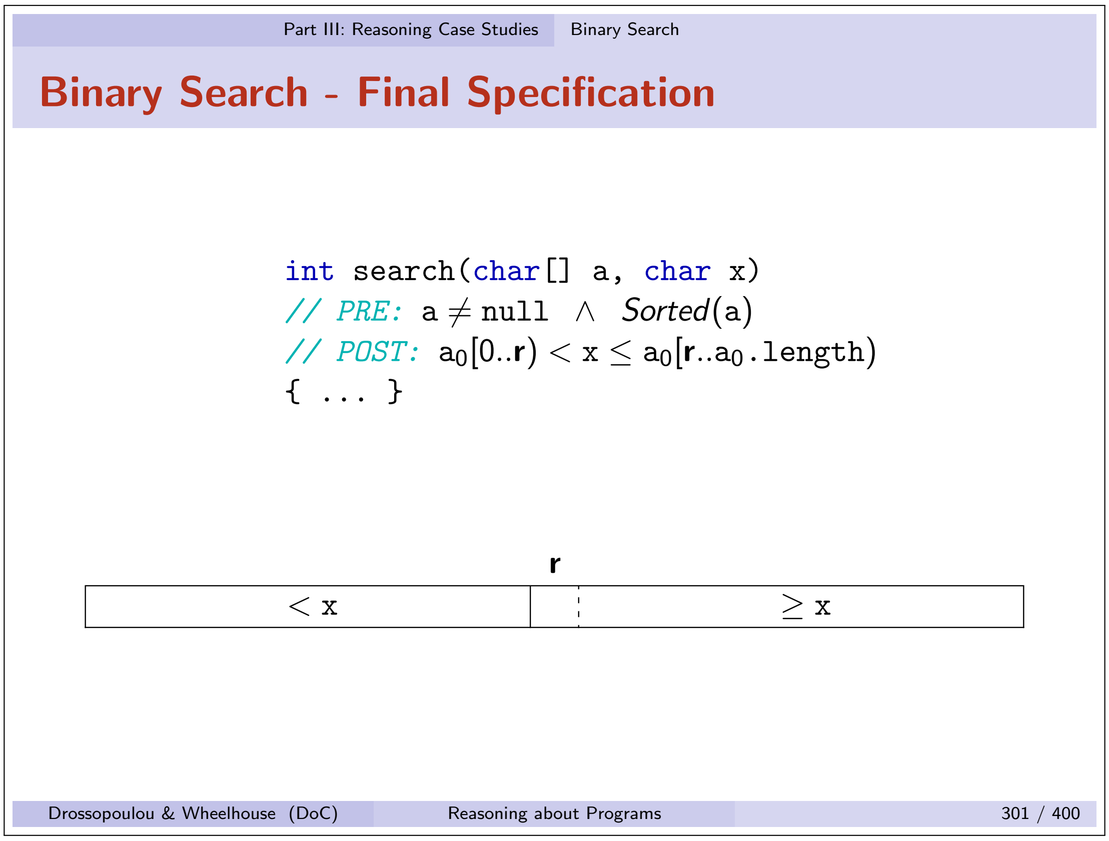
Notice that it we set $m \geq n$ then there will be no integers i in the range $m \leq i < n$ and the implication will hold vacuously.

Also notice that for values of i that are not in bounds of the array a, the assertion $0 \leq i < \texttt{a.length}$ will be false and once again the implication holds vacuously.

So, in $\texttt{a}[0..\mathbf{r}) < \texttt{x} \leq \texttt{a}[\mathbf{r}..\texttt{a.length})$ we can set $\mathbf{r}$ to values not in the range of a.

If we set $\mathbf{r} \leq 0$ then the first range is empty, so all elements of the array must be $\geq$ `x`.

If we instead set $\mathbf{r} \geq$ `a.length` then the second range is empty, so all elements of the array must be $\leq$ `x`.

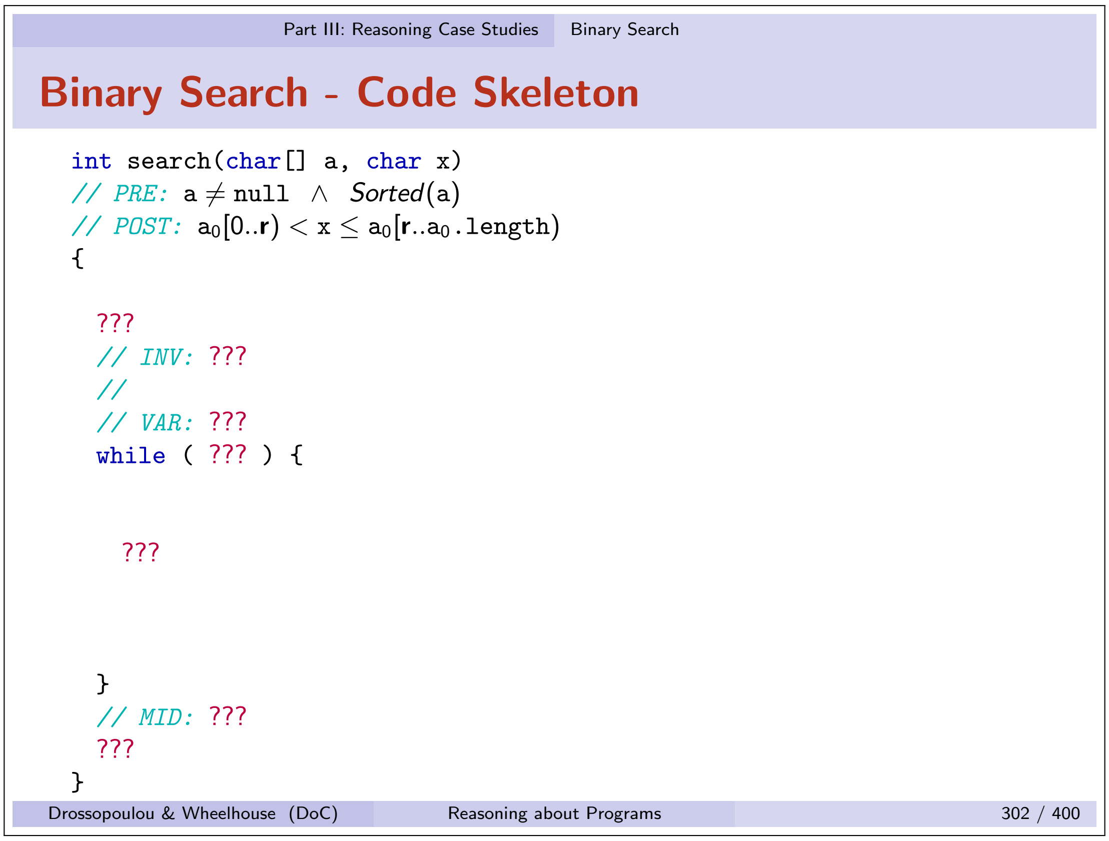


Binary Search - Final Specification

```
int search(char[] a, char x)
// PRE: a ≠ null  ∧  Sorted(a)
// POST: a₀[0..r) < x ≤ a₀[r..a₀.length)
{ ... }
```

Slide 301

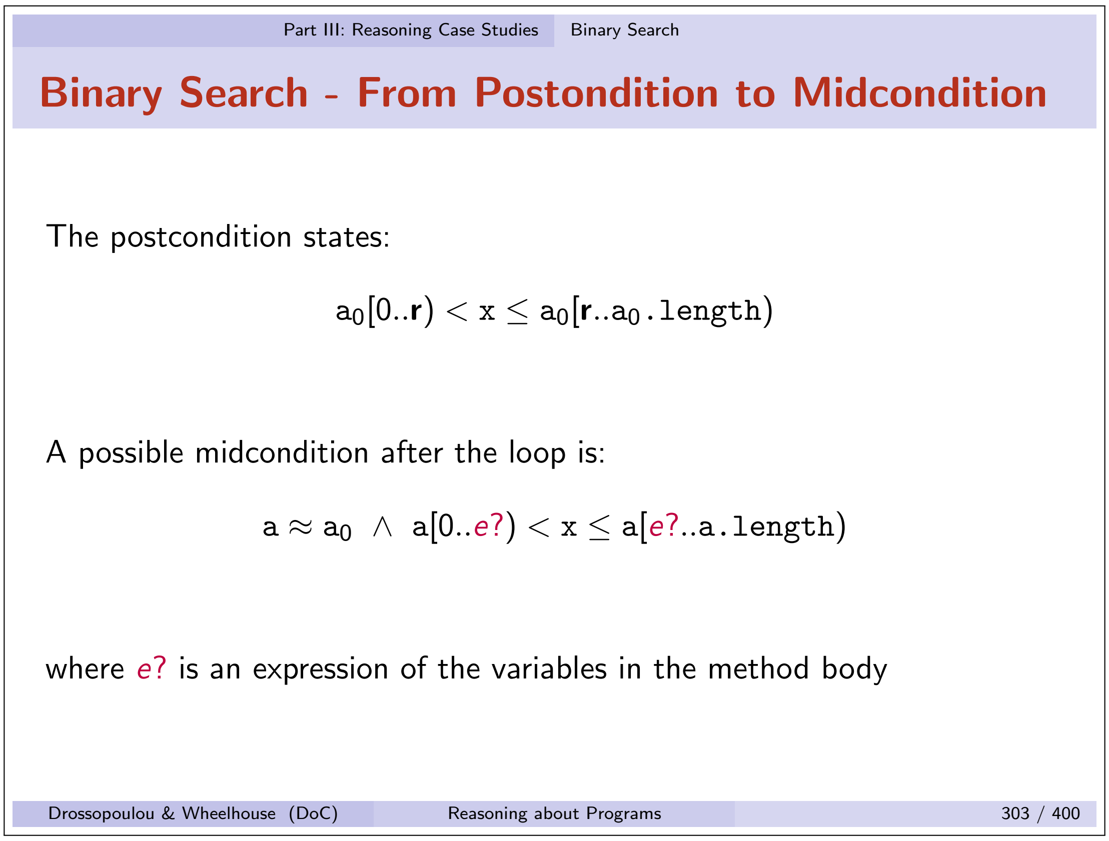
Slide 302



Binary Search - Code Skeleton

```
int search(char[] a, char x)
// PRE: a ≠ null  ∧  Sorted(a)
// POST: a₀[0..r) < x ≤ a₀[r..a₀.length)
{

  ???
  // INV: ???
  //
  // VAR: ???
  while ( ??? ) {


    ???




  }
  // MID: ???
  ???
}
```

Slide 303



Binary Search - From Postondition to Midcondition

The postcondition states:

$$\mathtt{a}_0[0..\mathbf{r}) < \mathtt{x} \leq \mathtt{a}_0[\mathbf{r}..\mathtt{a}_0\mathtt{.length})$$

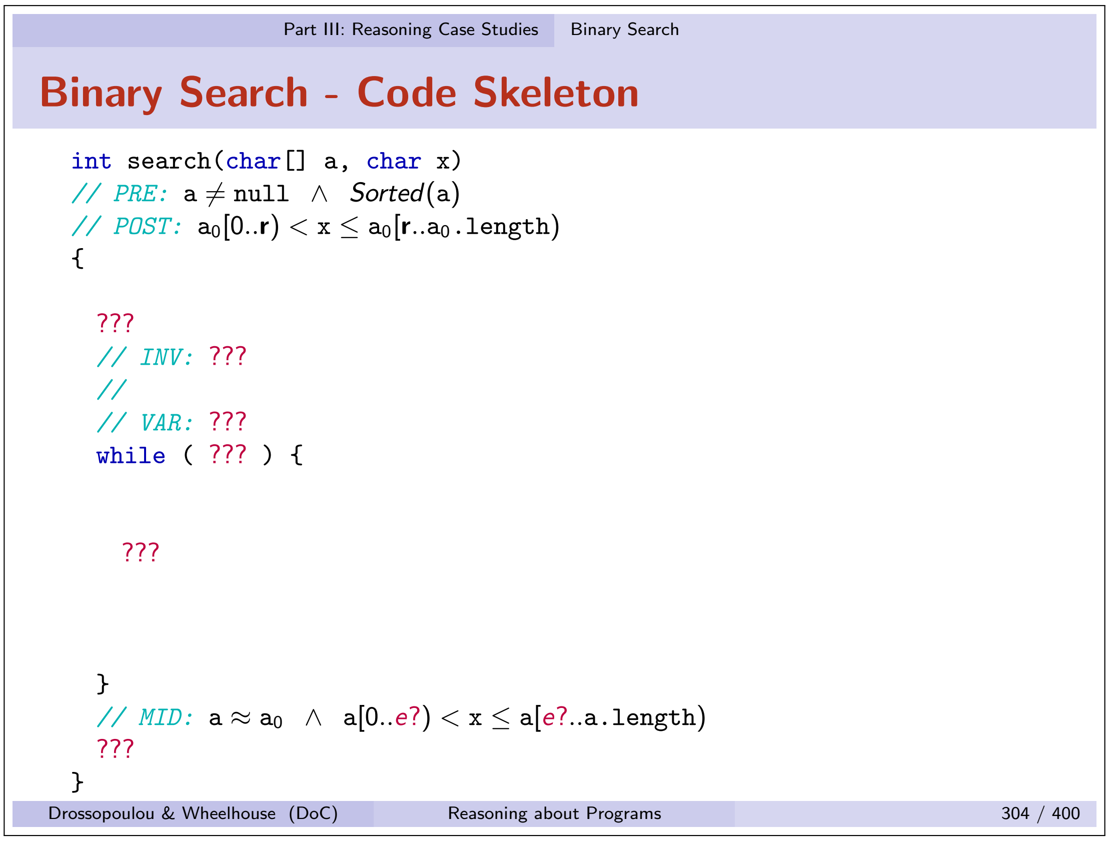
A possible midcondition after the loop is:

$$\mathtt{a} \approx \mathtt{a}_0 \;\wedge\; \mathtt{a}[0..e?) < \mathtt{x} \leq \mathtt{a}[e?..\mathtt{a.length})$$

where $e?$ is an expression of the variables in the method body

We will probably want the `search` method to return $e?$.

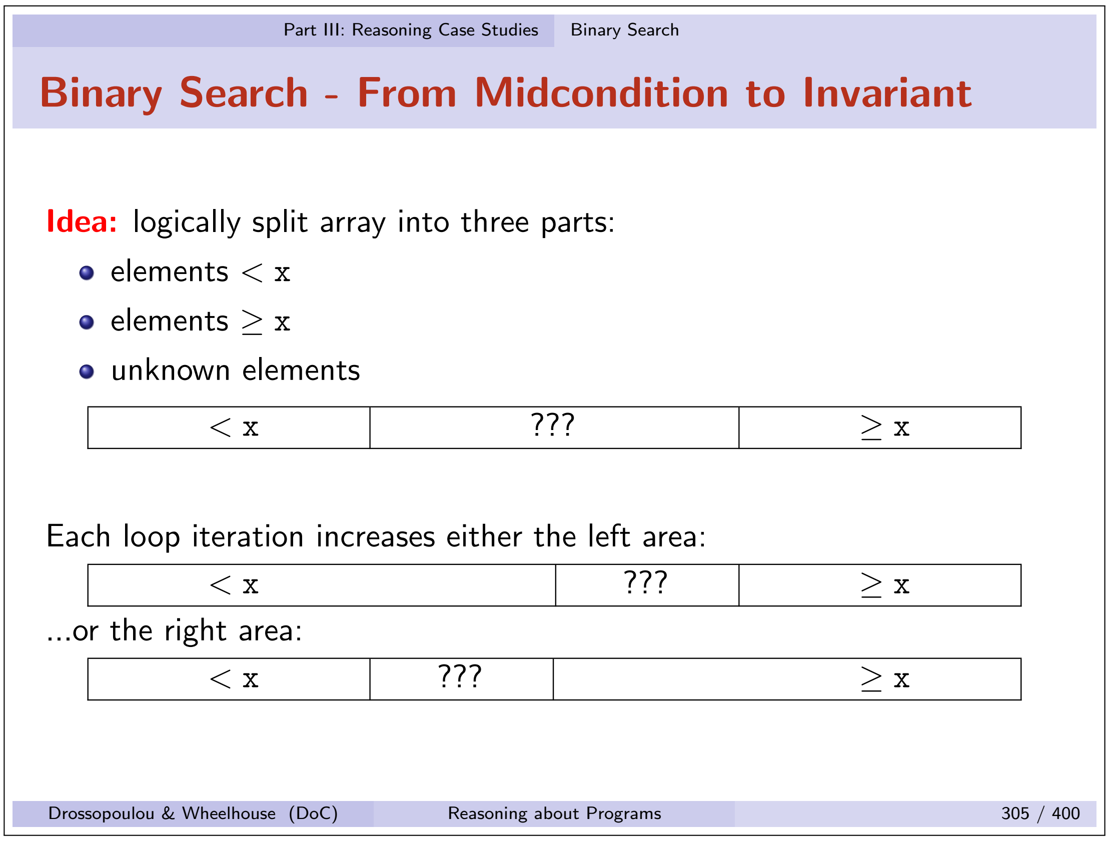
Slide 304



Binary Search - Code Skeleton

```
int search(char[] a, char x)
// PRE: a ≠ null  ∧  Sorted(a)
// POST: a0[0..r) < x ≤ a0[r..a0.length)
{

  ???
  // INV: ???
  //
  // VAR: ???
  while ( ??? ) {


    ???




  }
  // MID: a ≈ a0  ∧  a[0..e?) < x ≤ a[e?..a.length)
  ???
}
```

Slide 305



Binary Search - From Midcondition to Invariant

Idea: logically split array into three parts:

- elements $< \mathtt{x}$
- elements $\geq \mathtt{x}$
- unknown elements

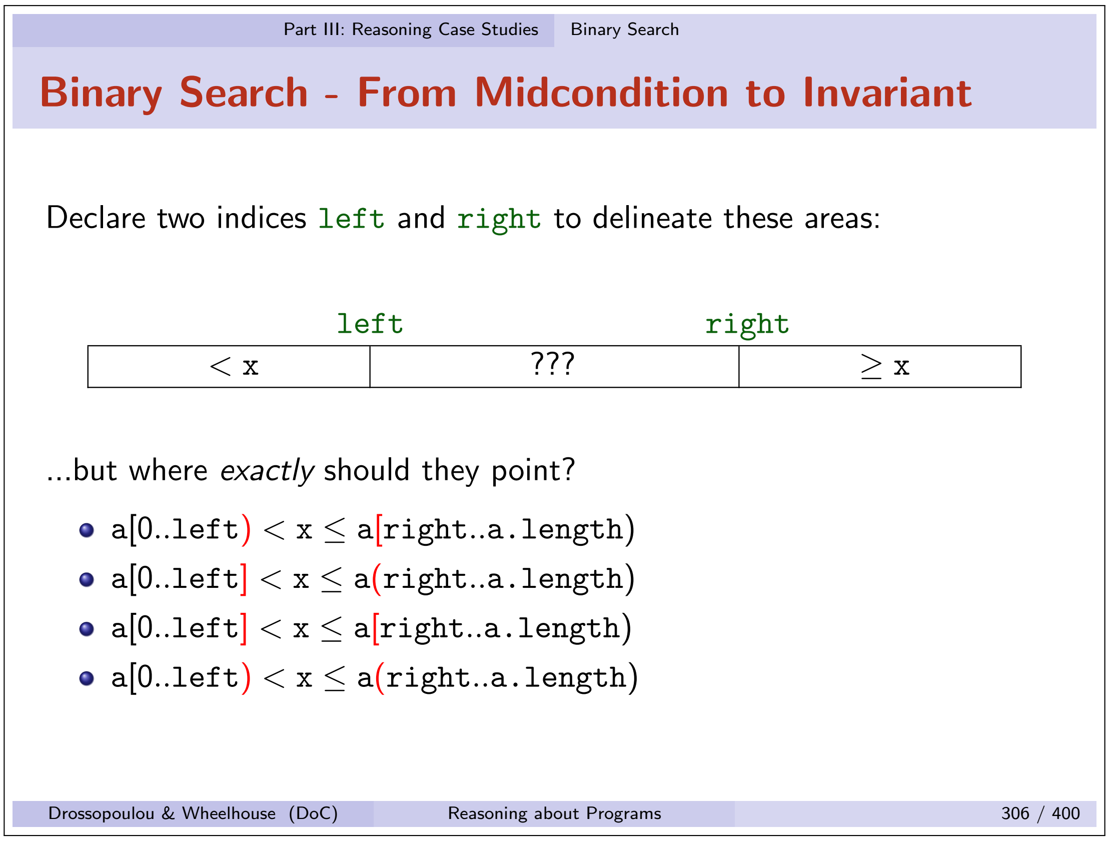
$< \mathtt{x}$	???	$\geq \mathtt{x}$

Each loop iteration increases either the left area:

$< \mathtt{x}$	???	$\geq \mathtt{x}$

...or the right area:

$< \mathtt{x}$	???	$\geq \mathtt{x}$



Binary Search - From Midcondition to Invariant

Declare two indices `left` and `right` to delineate these areas:

$<$ x	???	$\geq$ x

...but where *exactly* should they point?

- a[0..left) $<$ x $\leq$ a[right..a.length)
- a[0..left] $<$ x $\leq$ a(right..a.length)
- a[0..left] $<$ x $\leq$ a[right..a.length)
- a[0..left) $<$ x $\leq$ a(right..a.length)

There are four possible alternative for how to describe the areas of the array. How do we choose one?

Criteria for selecting an invariant:

- Can we write initialization code to establish invariant?
- Can we find a loop condition whose negation, together with the invariant, establishes the mid-condition?
- Can we make progress, and so re-establish the invariant?
- Can we find an appropriate variant (i.e. one which decreases in some well-founded ordering)?

Which alternative would you choose?

For the duration of our discussion we will arbitrarily choose to work with the 1st alternative, as this fits our desire to work with closed-open intervals whenever possible.

Interesting Question:

Does a[0..left) $<$ x $\leq$ a[right..a.length) imply that left $<$ right?

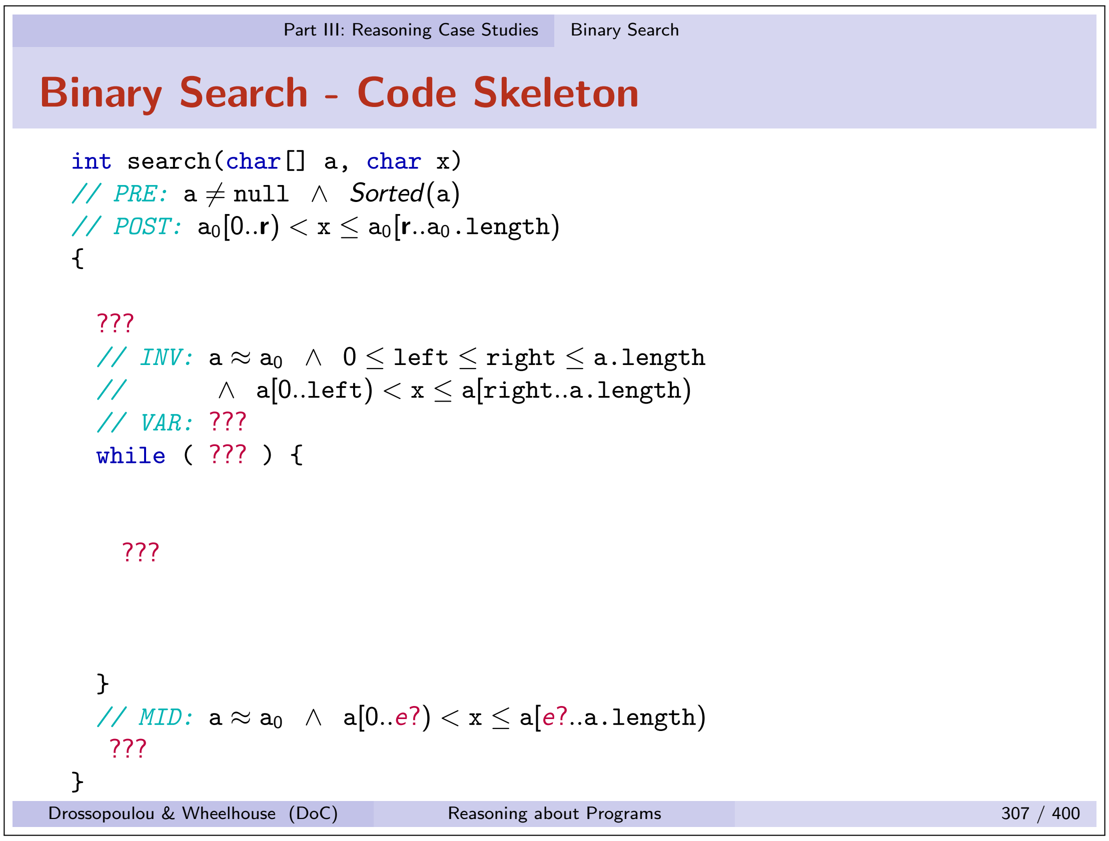
Answer:

No.

If there were some value i such that `a[i]` $<$ `x` and `a[i]` $\geq$ `x`, we would clearly have a problem. However, the bound on the end of the first array slice is open, while the bound on the start of the second array slice is closed. This means that `left` could be the same as `right` without causing such an issue.

Now, if `left` $>$ `right` and both were in the bounds of the array, then there *would* be a value in the overlap range and we would have the problem described above.

So, we choose to bound `left` and `right` to *sensible* values.



Binary Search - Code Skeleton

```
int search(char[] a, char x)
// PRE: a ≠ null  ∧  Sorted(a)
// POST: a₀[0..r) < x ≤ a₀[r..a₀.length)
{

  ???
  // INV: a ≈ a₀  ∧  0 ≤ left ≤ right ≤ a.length
  //        ∧  a[0..left) < x ≤ a[right..a.length)
  // VAR: ???
  while ( ??? ) {


    ???




  }
  // MID: a ≈ a₀  ∧  a[0..e?) < x ≤ a[e?..a.length)
   ???
}
```

Slide 308

Binary Search - From Invariant to Initialisation Code

Invariant:

$a \approx a_0 \land 0 \leq \texttt{left} \leq \texttt{right} \leq \texttt{a.length} \land a[0..\texttt{left}) < x \leq a[\texttt{right}..\texttt{a.length})$

The invariant can be vacuously established if:

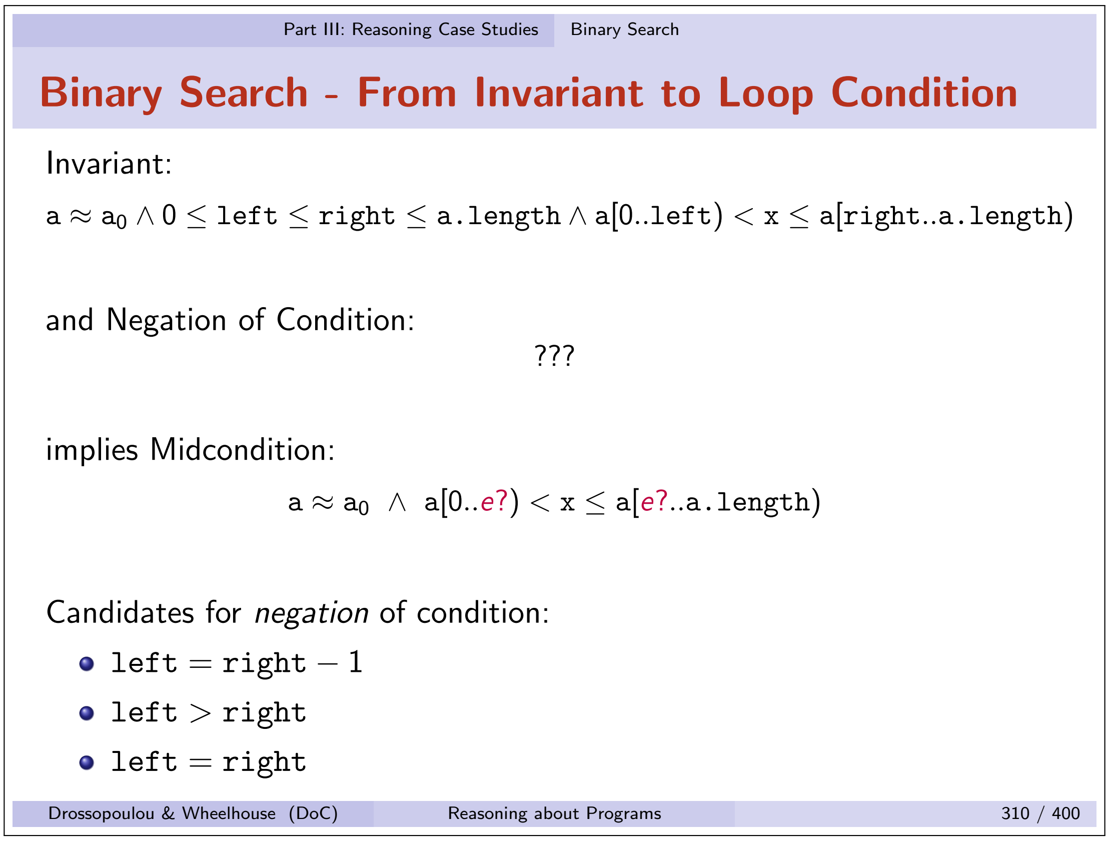
- $\texttt{left} = 0$
- $\texttt{right} = \texttt{a.length}$

So we should set this up with sensible initialisation code.

Slide 309

Binary Search - Code Skeleton

```
int search(char[] a, char x)
// PRE: a ≠ null  ∧  Sorted(a)
// POST: a₀[0..r) < x ≤ a₀[r..a₀.length)
{
  int left = 0;
  int right = a.length;
  // INV: a ≈ a₀  ∧  0 ≤ left ≤ right ≤ a.length
  //        ∧  a[0..left) < x ≤ a[right..a.length)
  // VAR: ???
  while ( ??? ) {


    ???



  }
  // MID: a ≈ a₀  ∧  a[0..e?) < x ≤ a[e?..a.length)
  ???
}
```



Binary Search - From Invariant to Loop Condition

Invariant:

$a \approx a_0 \land 0 \leq \texttt{left} \leq \texttt{right} \leq \texttt{a.length} \land \texttt{a}[0..\texttt{left}) < \texttt{x} \leq \texttt{a}[\texttt{right}..\texttt{a.length})$

and Negation of Condition:

???

implies Midcondition:

$$a \approx a_0 \ \land \ \texttt{a}[0..e?) < \texttt{x} \leq \texttt{a}[e?..\texttt{a.length})$$

Candidates for *negation* of condition:

- $\texttt{left} = \texttt{right} - 1$
- $\texttt{left} > \texttt{right}$
- $\texttt{left} = \texttt{right}$

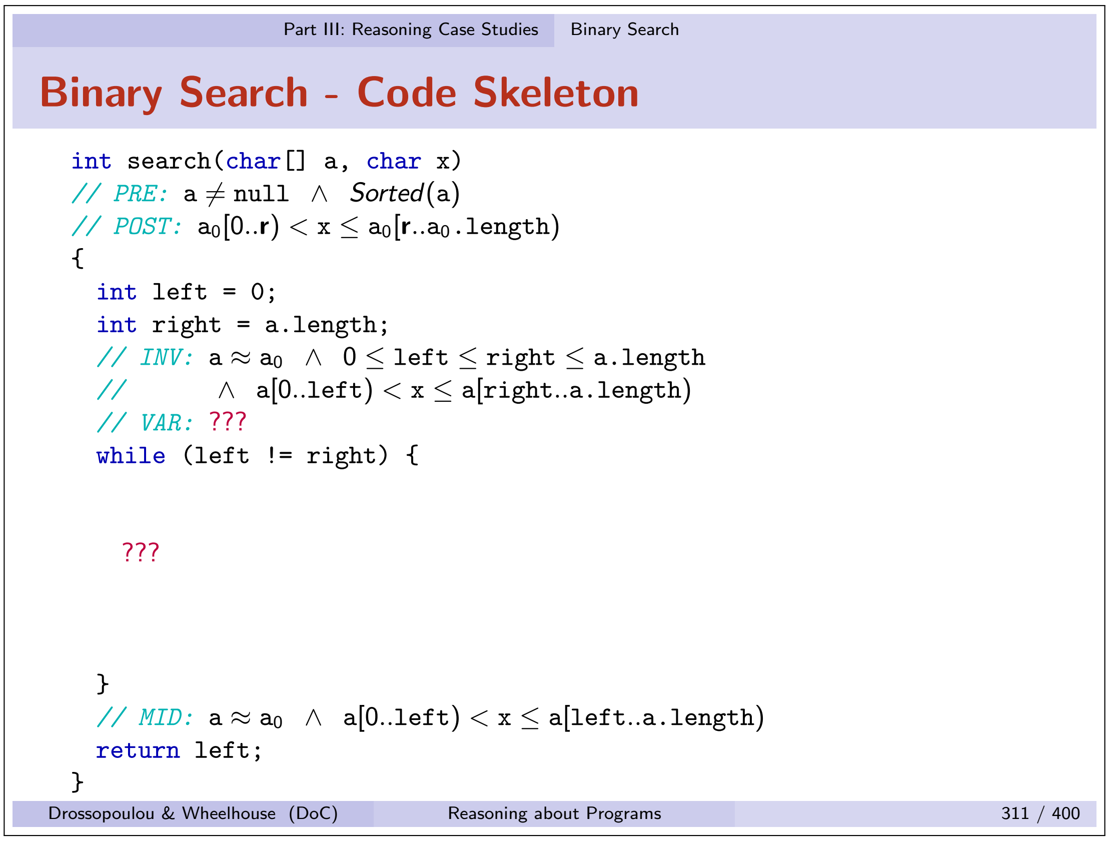
The choice of $\texttt{left} > \texttt{right}$ would invalidate the invariant.

The choice of $\texttt{left} = \texttt{right} - 1$ would leave an unchecked element in the array at position `left`, making it impossible to establish the postcondition.

The choice of $\texttt{left} = \texttt{right}$ does not have either of these problems, so we proceed with this. Of course, this means that our actual loop condition must establish that $\texttt{left} \neq \texttt{right}$. The simplest expression for this is `(left != right)`.

Note that we can now deduce that $e?$ can be either of `left` or `right`, so we pick `left` to save us a whole character of typing!

Slide 311



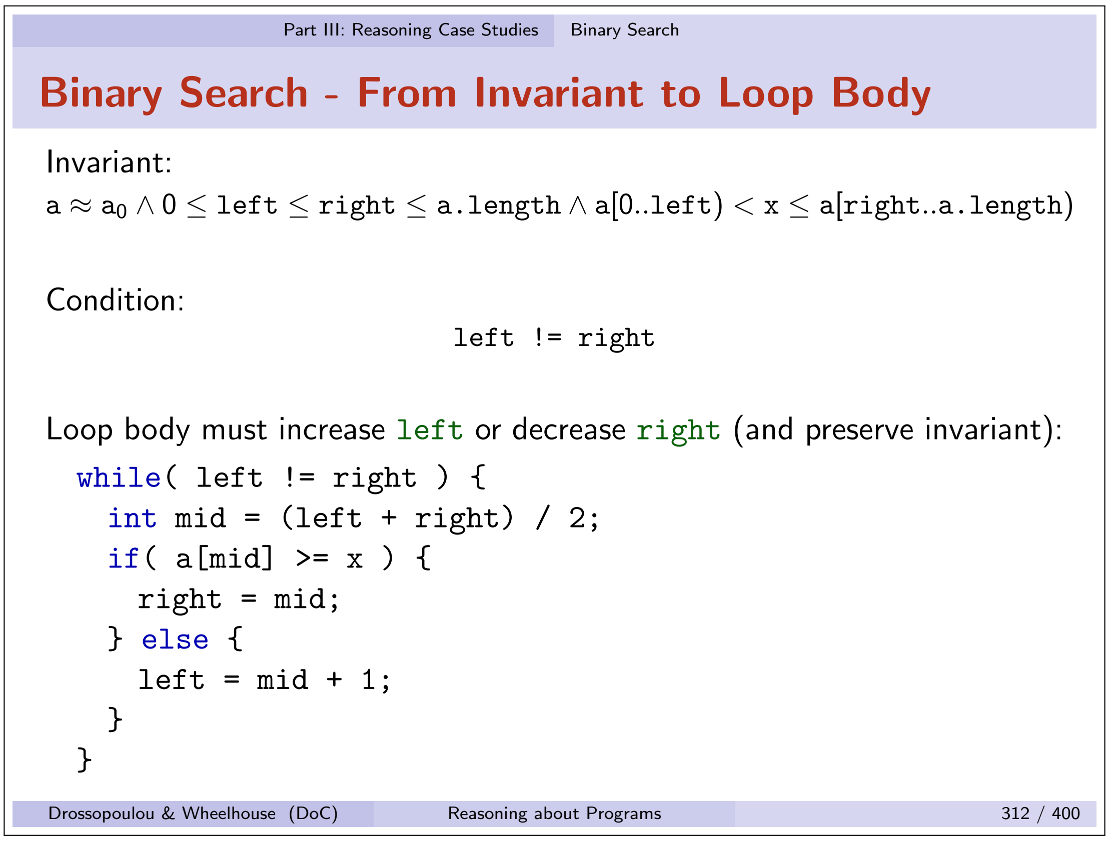
Part III: Reasoning Case Studies Binary Search

Binary Search - Code Skeleton

```
int search(char[] a, char x)
// PRE: a ≠ null  ∧  Sorted(a)
// POST: a₀[0..r) < x ≤ a₀[r..a₀.length)
{
  int left = 0;
  int right = a.length;
  // INV: a ≈ a₀  ∧  0 ≤ left ≤ right ≤ a.length
  //        ∧  a[0..left) < x ≤ a[right..a.length)
  // VAR: ???
  while (left != right) {


    ???



  }
  // MID: a ≈ a₀  ∧  a[0..left) < x ≤ a[left..a.length)
  return left;
}
```

Drossopoulou & Wheelhouse (DoC) Reasoning about Programs 311 / 400

Slide 312



Part III: Reasoning Case Studies Binary Search

Binary Search - From Invariant to Loop Body

Invariant:
$a \approx a_0 \land 0 \leq \texttt{left} \leq \texttt{right} \leq \texttt{a.length} \land a[0..\texttt{left}) < x \leq a[\texttt{right}..\texttt{a.length})$

Condition:

`left != right`

Loop body must increase `left` or decrease `right` (and preserve invariant):

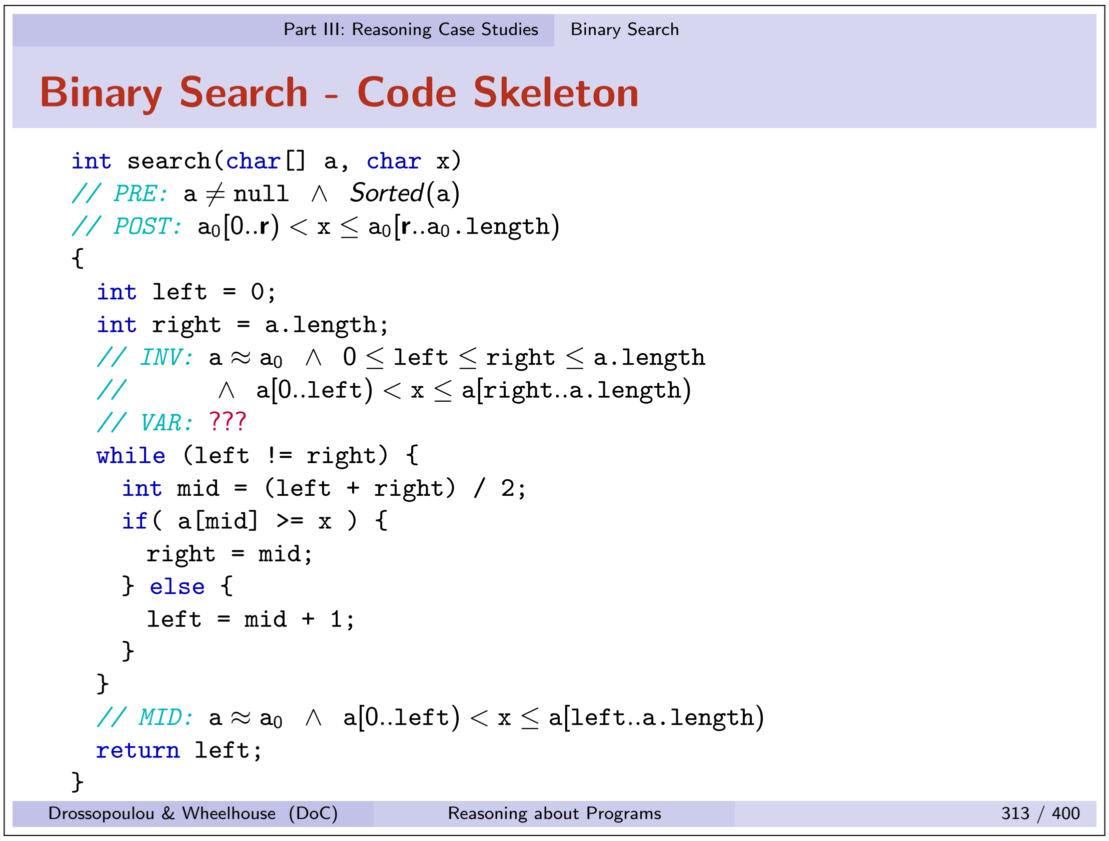
```
while( left != right ) {
  int mid = (left + right) / 2;
  if( a[mid] >= x ) {
    right = mid;
  } else {
    left = mid + 1;
  }
}
```

Drossopoulou & Wheelhouse (DoC) Reasoning about Programs 312 / 400

Whilst this might seem like a lot of code to pluck out of this air, its development is pretty straightforward. We already had the rough structure of the code from our pseudo code for the algorithm.

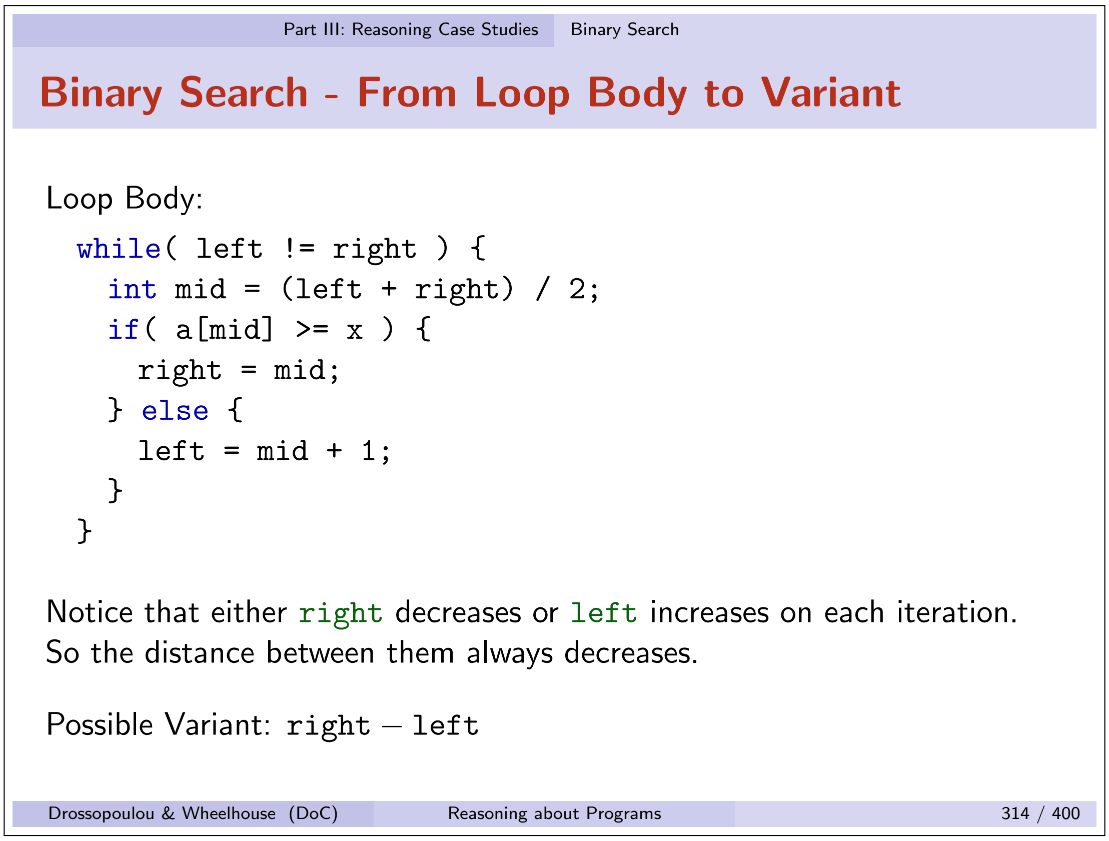
The first step is to choose how to pick the mid-point. Then we test the element at this mid-point. Finally, we adjust the bounds `left` or `right` to reflect the new observation we have made.

Note, in particular, that the adjustment of the `left` range must include a +1 offset. This is due to the range a[0..`left`) being open on its right-hand side. So the element at position `left` has not yet actually been inspected.



Binary Search - Code Skeleton

```
int search(char[] a, char x)
// PRE: a ≠ null  ∧  Sorted(a)
// POST: a₀[0..r) < x ≤ a₀[r..a₀.length)
{
  int left = 0;
  int right = a.length;
  // INV: a ≈ a₀  ∧  0 ≤ left ≤ right ≤ a.length
  //        ∧  a[0..left) < x ≤ a[right..a.length)
  // VAR: ???
  while (left != right) {
    int mid = (left + right) / 2;
    if( a[mid] >= x ) {
      right = mid;
    } else {
      left = mid + 1;
    }
  }
  // MID: a ≈ a₀  ∧  a[0..left) < x ≤ a[left..a.length)
  return left;
}
```



Binary Search - From Loop Body to Variant

Loop Body:

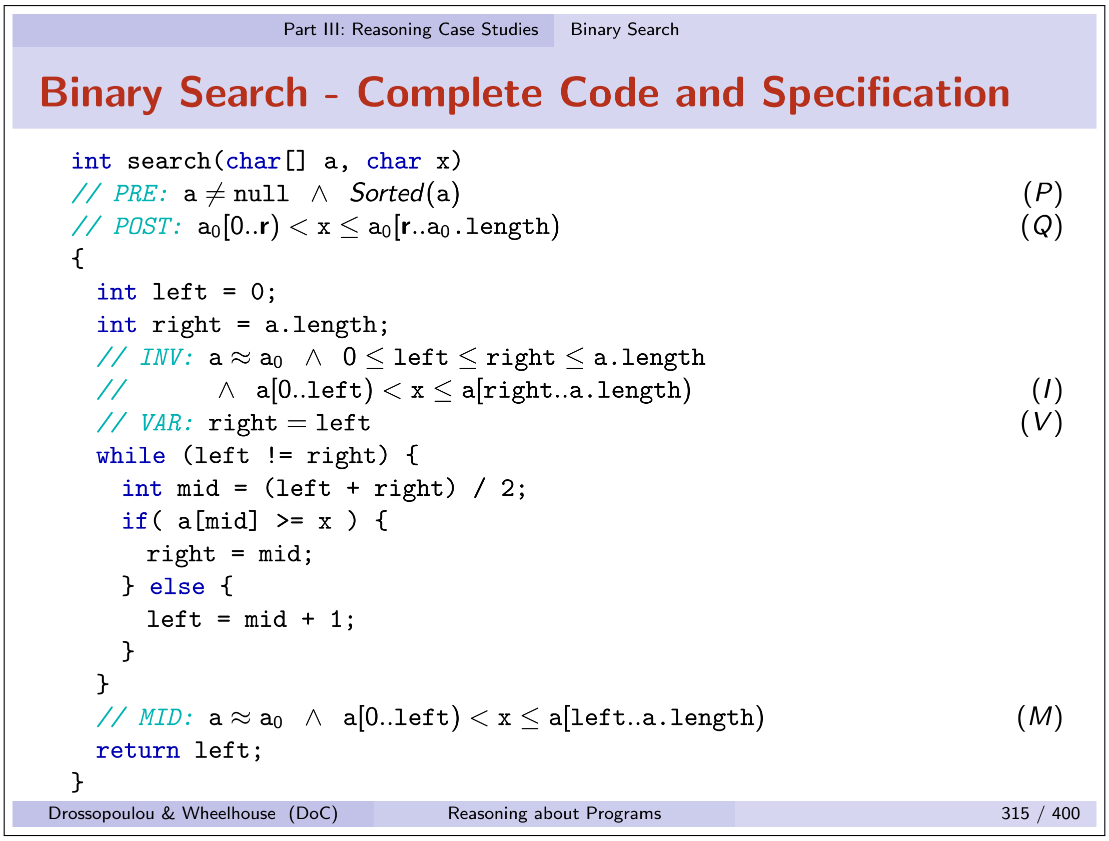
```
while( left != right ) {
  int mid = (left + right) / 2;
  if( a[mid] >= x ) {
    right = mid;
  } else {
    left = mid + 1;
  }
}
```

Notice that either `right` decreases or `left` increases on each iteration.
So the distance between them always decreases.

Possible Variant: `right` − `left`

Note that neither −`left` or `right` would be suitable variants, as they do not decrease on *every* loop iteration. You could choose to define an ordering on the pair (`left`, `right`), but this will boil down to the same thing as `right` − `left`, so let's not overcomplicate things.

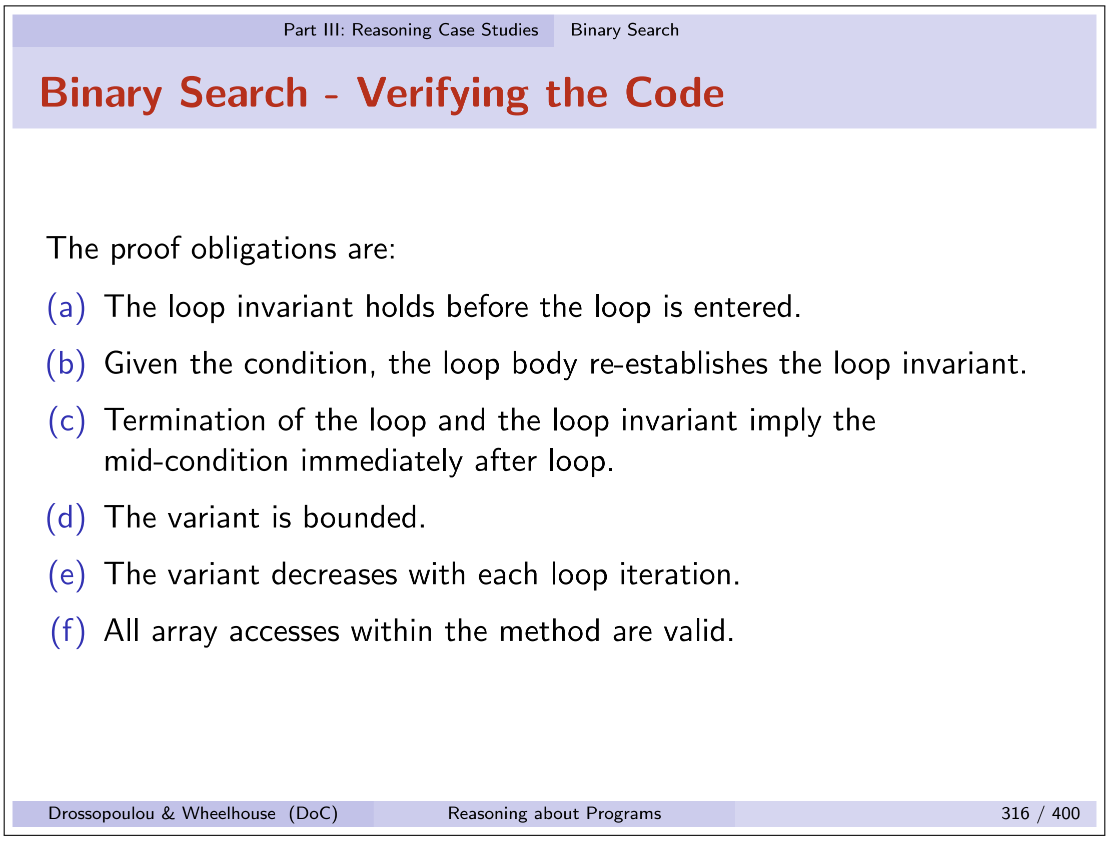
Slide 315



Binary Search - Complete Code and Specification

```
int search(char[] a, char x)
// PRE: $a \neq null \ \wedge \ Sorted(a)$                                    $(P)$
// POST: $a_0[0..\mathbf{r}) < x \leq a_0[\mathbf{r}..a_0.length)$                          $(Q)$
{
  int left = 0;
  int right = a.length;
  // INV: $a \approx a_0 \ \wedge \ 0 \leq left \leq right \leq a.length$
  //      $\wedge \ a[0..left) < x \leq a[right..a.length)$                     $(I)$
  // VAR: $right - left$                                                  $(V)$
  while (left != right) {
    int mid = (left + right) / 2;
    if( a[mid] >= x ) {
      right = mid;
    } else {
      left = mid + 1;
    }
  }
  // MID: $a \approx a_0 \ \wedge \ a[0..left) < x \leq a[left..a.length)$               $(M)$
  return left;
}
```

Slide 316



Binary Search - Verifying the Code

The proof obligations are:

(a) The loop invariant holds before the loop is entered.

(b) Given the condition, the loop body re-establishes the loop invariant.

(c) Termination of the loop and the loop invariant imply the mid-condition immediately after loop.

(d) The variant is bounded.

(e) The variant decreases with each loop iteration.

(f) All array accesses within the method are valid.

By the code construction technique we have employed, these proofs ought to be pretty straightforward. However, we should still check them, just to be sure that we have not made any mistakes.

(a) Invariant holds before the loop is entered

$$\boxed{\begin{array}{c} P[\mathsf{a} \mapsto \mathsf{a}_0] \ \wedge \ \texttt{left} = 0 \ \wedge \ \texttt{right} = \texttt{a.length} \ \wedge \ \mathsf{a} \approx \mathsf{a}_0 \\ \longrightarrow \\ I \end{array}}$$

(b) Loop body re-establishes the loop invariant

$$\boxed{\begin{array}{c} I \ \wedge \ \mathsf{a}' \approx \mathsf{a} \ \wedge \ \texttt{left} \neq \texttt{right} \ \wedge \ \texttt{mid} = (\texttt{left} + \texttt{right})/2 \\ \wedge \ (\mathsf{a}\texttt{[mid]} \geq \texttt{x} \longrightarrow \texttt{left}' = \texttt{left} \wedge \texttt{right}' = \texttt{mid}) \\ \vee \ (\mathsf{a}\texttt{[mid]} < \texttt{x} \longrightarrow \texttt{left}' = \texttt{mid} + 1 \wedge \texttt{right}' = \texttt{right}) \\ \longrightarrow \\ I[\mathsf{a} \mapsto \mathsf{a}', \texttt{left} \mapsto \texttt{left}', \texttt{right} \mapsto \texttt{right}'] \end{array}}$$

(c) Midcondition holds straight after loop

$$\boxed{\begin{array}{c} I \ \wedge \ \texttt{left} = \texttt{right} \\ \longrightarrow \\ M \end{array}}$$

(d) + (e) The loop terminates

$$\boxed{\begin{array}{c} I \ \wedge \ \mathsf{a}' \approx \mathsf{a} \ \wedge \ \texttt{left} \neq \texttt{right} \ \wedge \ \texttt{mid} = (\texttt{left} + \texttt{right})/2 \\ \wedge \ (\mathsf{a}\texttt{[mid]} \geq \texttt{x} \longrightarrow \texttt{left}' = \texttt{left} \wedge \texttt{right}' = \texttt{mid}) \\ \vee \ (\mathsf{a}\texttt{[mid]} < \texttt{x} \longrightarrow \texttt{left}' = \texttt{mid} + 1 \wedge \texttt{right}' = \texttt{right}) \\ \longrightarrow \\ V \text{ bound} \ \wedge \ V[\mathsf{a} \mapsto \mathsf{a}', \texttt{left} \mapsto \texttt{left}', \texttt{right} \mapsto \texttt{right}'] < V \end{array}}$$

(f) Array access are legal

$$\boxed{\begin{array}{c} I \ \wedge \ \texttt{left} \neq \texttt{right} \ \wedge \ \texttt{mid} = (\texttt{left} + \texttt{right})/2 \\ \longrightarrow \\ 0 \leq \texttt{mid} < \texttt{a.length} \end{array}}$$

These proofs are left as an exercise for the reader.