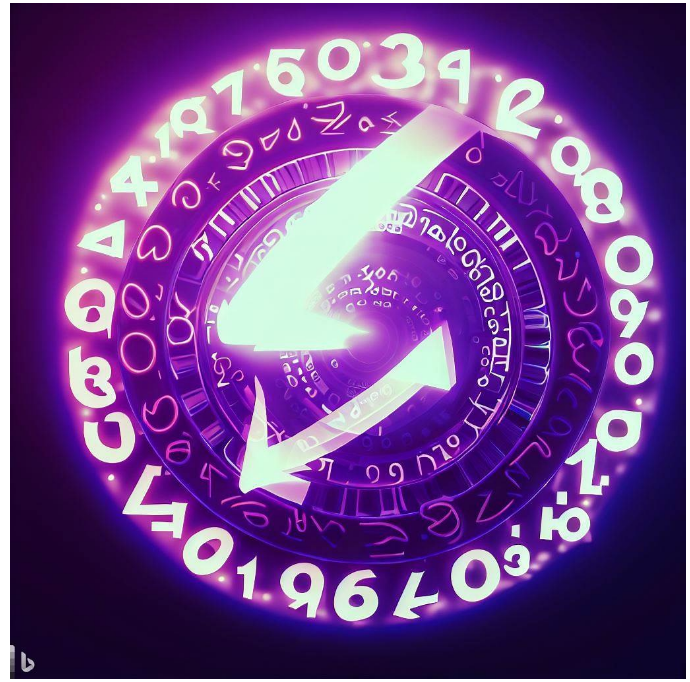


CSE 331

Array Loop Heuristics

Kevin Zatloukal

Recall: Sum of an Array

$$\textbf{func}\ \mathrm{sum}([]) := 0$$
$$\mathrm{sum}(A +\!\!+ [y]) := \mathrm{sum}(A) + y \qquad \text{for any } y : \mathbb{Z} \text{ and } A : \mathrm{Array}_{\mathbb{Z}}$$

- **Loop implementation:**

```
let j: number = 0;
let s: number = 0;
{{ Inv: s = sum(A[0 .. j – 1]) and 0 ≤ j ≤ A.length }}
while (j !== A.length) {
  s = s + A[j];
  j = j + 1;
}
{{ s = sum(A) }}
return s;
```

Recall: Linear Search of an Array

func contains([], x) := F
  contains(A ⧺ [y], x) := T if x = y
  contains(A ⧺ [y], x) := contains(A, x) if x ≠ y

- **Loop implementation:**

```
let j: number = 0;
{{ Inv: contains(A[0 .. j−1], x) = F and 0 ≤ j ≤ A.length }}
while (j != A.length) {
  if (A[j] === x)
    {{ contains(A, x) = T }}
    return true;
  j = j + 1;
}
{{ contains(A, x) = F }}
return false;
```

Recall: Linear Search of an Array (Loop Body)

func contains([], x) := F
    contains(A ⧺ [y], x) := T   if x = y
    contains(A ⧺ [y], x) := contains(A, x)   if x ≠ y

- **Loop implementation:**

```
{{ contains(A[0 .. j–1], x) = F and 0 ≤ j < A.length }}
if (A[j] === x) {
  {{ contains(A, x) = T }}
  return true;
} else {

}
j = j + 1;
{{ contains(A[0 .. j–1], x) = F and 0 ≤ j ≤ A.length }}
```

Loop Invariants with Arrays

- **Saw two more examples last lecture**

{{ **Inv**: $s = \text{sum}(A[0 .. j-1])$ … }} **sum of array**
{{ **Post**: $s = \text{sum}(A[0 .. n-1])$ }}

{{ **Inv**: $\text{contains}(A[0 .. j-1], x) = F$ … }} **search an array**
{{ **Post**: $\text{contains}(A[0 .. n-1], x) = F$ }}

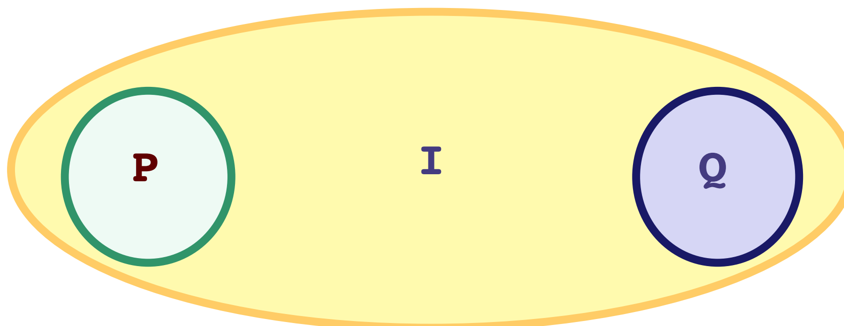
  – **in both cases,** Post **is a special case of** Inv **(where** $j = n$**)**
  – **in other words,** Inv **is a weakening of** Post

- **Heuristic for loop invariants: weaken the postcondition**
  – **assertion that allows postcondition as a special case**
  – **must also allow states that are easy to prepare**

Heuristic for Loop Invariants

- Loop Invariant allows both start and stop states
  - describing more states = weakening

```
{{ P }}
{{ Inv: I }}
while (cond) {
  S
}
{{ Q }}
```



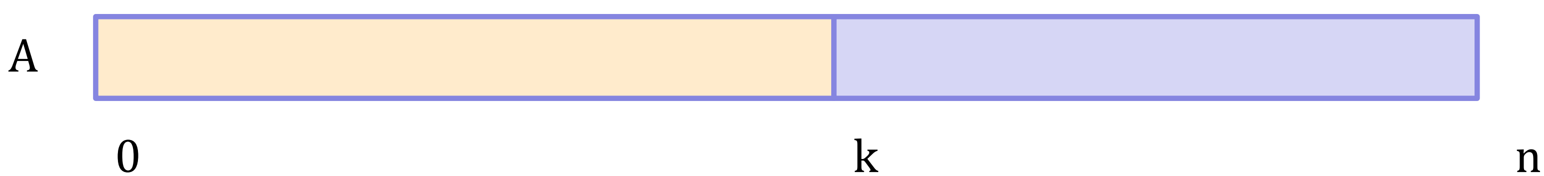
  - usually are many ways to weaken it...

Searching a Sorted Array

- **Suppose we require A to be sorted:**
  - **precondition includes**

$A[j-1] \leq A[j]$ for any $1 \leq j < n$ (**where** $n := A.\text{length}$)

- **Want to find the index k where "x" would be...**



$A[j] < x$ for any $0 \leq j < k$ and $x \leq A[j]$ for any $k \leq j < n$

Searching a Sorted Array

- Suppose we require A to be sorted:
  - precondition includes

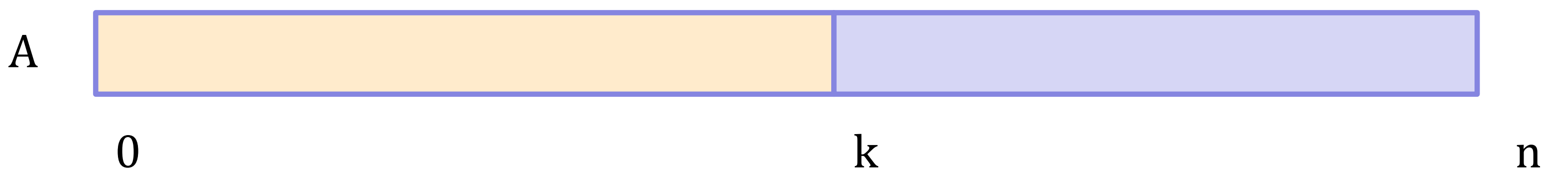
    $A[j-1] \leq A[j]$ for any $1 \leq j < n$ (where $n := A.length$)

- Want to find the index k where "x" would be
  - loop postcondition written as

    $A[j] < x$ for any $0 \leq j \leq k-1$ and $x \leq A[j]$ for any $k \leq j < n$

  - index k is where x must be if it is present

Searching a Sorted Array

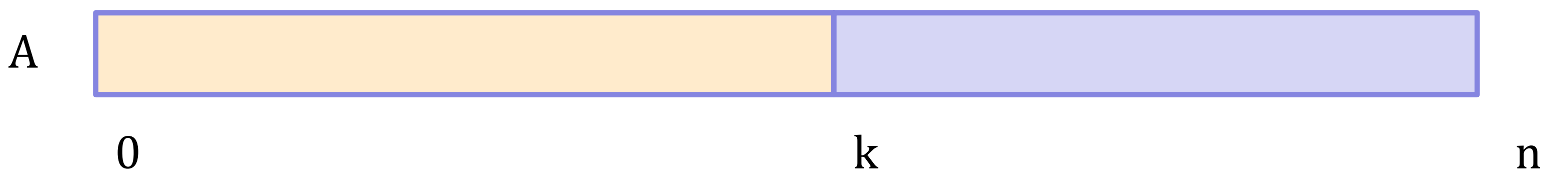


$A[j] < x$ for any $0 \leq j < k$ and $x \leq A[j]$ for any $k \leq j < n$

- **End with complete knowledge of $A[j]$ vs x**
  - **how can we describe *partial* knowledge?**

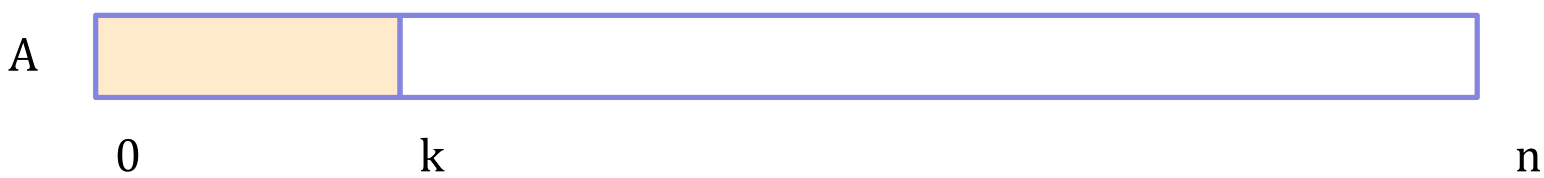
- **Recall: loop for** contains
  - **postcondition says to return** $\text{contains}(A, x)$
  - **but we exit loop knowing** $\text{contains}(A, x) = F$

Searching a Sorted Array



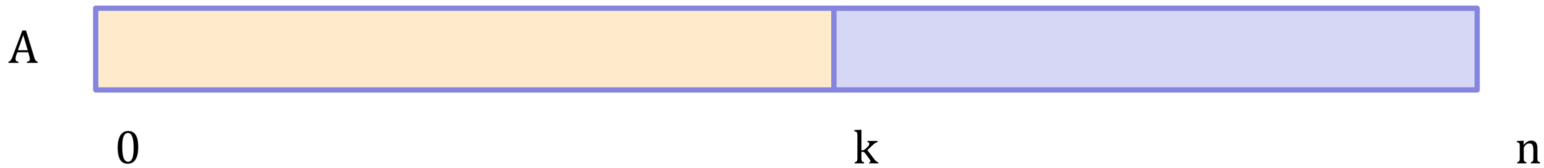
$A[j] < x$ for any $0 \leq j < k$ and $x \leq A[j]$ for any $k \leq j < n$

- End with complete knowledge of $A[j]$ vs x
  - how can we describe *partial* knowledge?
  - we will focus on the elements that are smaller than x



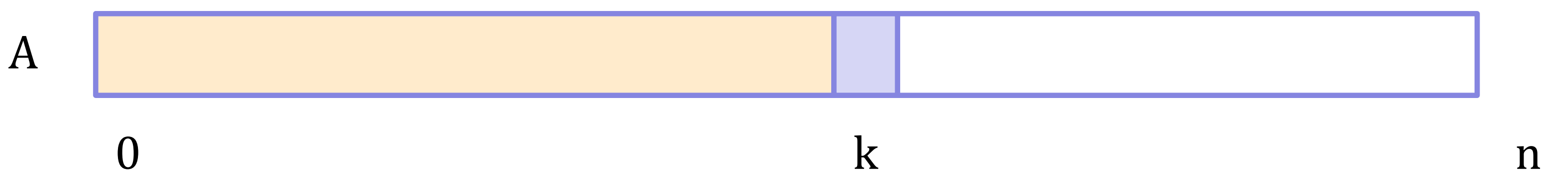
$A[j] < x$ for any $0 \leq j < k$

Searching a Sorted Array



$A[j] < x$ for any $0 \le j < k$ and $x \le A[j]$ for any $k \le j < n$

- **End with complete knowledge of $A[j]$ vs x**
  - **how can we describe *partial* knowledge?**



- **Loop idea… increase k until we hit $x \le A[k]$**

Searching a Sorted Array

```
// @returns true if A[j] = x for some 0 <= j < n
//          false if A[j] != x for any 0 <= j < n
```

- **Loop implementation:**

```
let k: number = 0;
{{ Inv: A[j] < x for any 0 ≤ j < k and 0 ≤ k ≤ n }}
while (k < A.length && A[k] <= x) {
  if (A[k] === x) {
    return true;
  } else {
    k = k + 1;
  }
}
return false;
```

Searching a Sorted Array

```
let k: number = 0;
{{ k = 0 }}
{{ Inv: A[j] < x for any 0 ≤ j < k and 0 ≤ k ≤ n }}
while (k < A.length && A[k] <= x) {
  if (A[k] === x) {
    return true;
  } else {
    k = k + 1;
  }
}
return false;
```

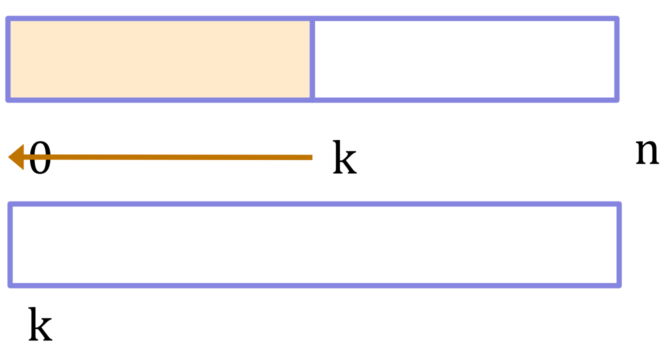
What is the claim when $k = 0$?

$A[j] < x$ for any $0 \le j < 0$

What values of j satisfy $0 \le j < 0$?

None. Nothing is claimed.

Statement is (vacuously) true when $k = 0$



With "for any" facts, we need to think about exactly what facts are being claimed.

Searching a Sorted Array

```
let k: number = 0;
{{ Inv: A[j] < x for any 0 ≤ j < k and 0 ≤ k ≤ n }}
while (k < A.length && A[k] <= x) {
  if (A[k] === x) {
    return true;
  } else {
    k = k + 1;
  }
}
{{ A[j] < x for any 0 ≤ j < k and (k = n or A[k] > x) }}
{{ A[j] ≠ x for any 0 ≤ j < n }}
return false;
```

Top assertion has an "or", so we argue by cases.

Searching a Sorted Array

```
while (k < A.length && A[k] <= x) {
  if (A[k] === x) {
    return true;
  } else {
    k = k + 1;
  }
}
{{ A[j] < x for any 0 ≤ j < k and (k = n or A[k] > x) }}
{{ A[j] ≠ x for any 0 ≤ j < n }}
return false;
```

Case $k = n$ ($= A.length$):

Know that $A[j] < x$ for any $0 \leq j < n$ (**since** $k = n$)

This means $A[j] \neq x$ for any $0 \leq j < n$ (**since** $A[j] < x$ implies $A[j] \neq x$)

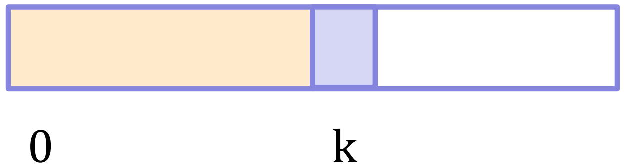
Searching a Sorted Array

```
while (k < A.length && A[k] <= x) {
  if (A[k] === x) {
    return true;
  } else {
    k = k + 1;
  }
}
```

{{ $A[j] < x$ for any $0 \le j < k$ and ($k = n$ or $A[k] > x$) }}

{{ $A[j] \ne x$ for any $0 \le j < n$ }}

```
return false;
```



Case $x < A[k]$:

Know that $A[j] < x$ for any $0 \le j < k$ and $x < A[k]$

Precondition (sorted) says $A[k] \le A[k+1] \le \dots$

Know that $A[j] < x$ for any $0 \le j < k$ and $x < A[j]$ for any $k \le j < n$

This means $A[j] \ne x$ for any $0 \le j < n$

Searching a Sorted Array

```
while (k < A.length && A[k] <= x) {
  if (A[k] === x) {
    return true;
  } else {
    k = k + 1;
  }
}
```

{{ $A[j] < x$ for any $0 \leq j < k$ and ($k = n$ or $A[k] > x$) }}

{{ $A[j] \neq x$ for any $0 \leq j < n$ }}

```
return false;
```

Since one of the cases $k = n$ and $x < A[k]$ must hold, we have shown that

$$A[j] \neq x \text{ for any } 0 \leq j < n$$

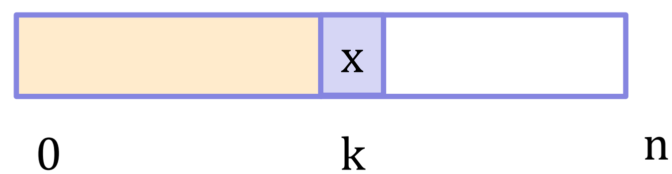
holds in general.

Searching a Sorted Array

```
let k: number = 0;
{{ Inv: A[j] < x for any 0 ≤ j < k and 0 ≤ k ≤ n }}
while (k < A.length && A[k] <= x) {
  {{ A[j] < x for any 0 ≤ j < k and 0 ≤ k < n and A[k] ≤ x }}
  if (A[k] === x) {
    return true;
  } else {
    k = k + 1;
  }
  {{ A[j] < x for any 0 ≤ j < k and 0 ≤ k ≤ n }}
}
return false;
```

Searching a Sorted Array

```
{{ Inv: A[j] < x for any 0 ≤ j < k and 0 ≤ k ≤ n }}
while (k < A.length && A[k] <= x) {
  {{ A[j] < x for any 0 ≤ j < k and 0 ≤ k < n and A[k] ≤ x }}
  if (A[k] === x) {
    {{ A[j] < x for any 0 ≤ j < k and 0 ≤ k < n and A[k] = x }}
    {{ A[j] = x for some 0 ≤ j < n }}
    return true;
  }
```

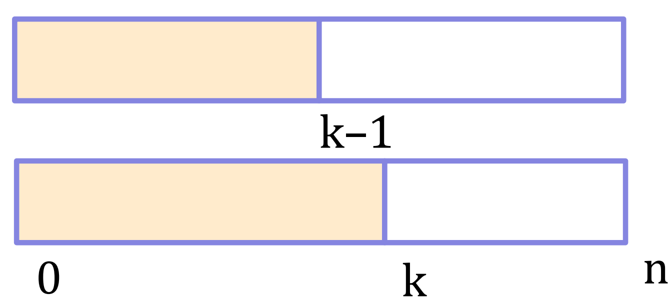


Is the postcondition true?

Yes! It holds for $j = k$

Searching a Sorted Array

```
{{ Inv: A[j] < x for any 0 ≤ j < k and 0 ≤ k ≤ n }}
while (k < A.length && A[k] <= x) {
  {{ A[j] < x for any 0 ≤ j < k and 0 ≤ k < n and A[k] ≤ x }}
  if (A[k] === x) {
    return true;
  } else {
    {{ A[j] < x for any 0 ≤ j < k and 0 ≤ k < n and A[k] < x }}
    k = k + 1;
    {{ A[j] < x for any 0 ≤ j < k-1 and 0 ≤ k-1 < n and A[k-1] < x }}
  }
  {{ A[j] < x for any 0 ≤ j < k-1 and 0 ≤ k-1 < n and A[k-1] < x }}
  {{ A[j] < x for any 0 ≤ j < k and 0 ≤ k ≤ n }}
}
return false;
```



Step 1: What facts need proof?

Only A[k-1] < x

Searching a Sorted Array

```
{{ Inv: A[j] < x for any 0 ≤ j < k and 0 ≤ k ≤ n }}
while (k < A.length && A[k] <= x) {
  {{ A[j] < x for any 0 ≤ j < k and 0 ≤ k < n and A[k] ≤ x }}
  if (A[k] === x) {
    return true;
  } else {
    {{ A[j] < x for any 0 ≤ j < k and 0 ≤ k < n and A[k] < x }}
    k = k + 1;
    {{ A[j] < x for any 0 ≤ j < k-1 and 0 ≤ k-1 < n and A[k-1] < x }}
  }
  {{ A[j] < x for any 0 ≤ j < k-1 and 0 ≤ k-1 < n and A[k-1] < x }}
  {{ A[j] < x for any 0 ≤ j < k and 0 ≤ k ≤ n }}
}
return false;
```

Step 1: What facts need proof?

Only $A[k-1] < x$

Step 2: prove the new fact(s)

$A[k-1] < x$ is known

Loops Invariants with Arrays

- Loop invariants often have *lots* of facts
  - recursion has fewer

- Much of the work is just keeping track of them
  - "dynamic programs" (421) are often like this
  - common to need to write these down

    more likely to see line-by-line reasoning on hard problems

Loops Invariants with Arrays

Implications btw "for any" facts are proven in two steps:

1. **Figure out what facts are <u>not</u> already known**

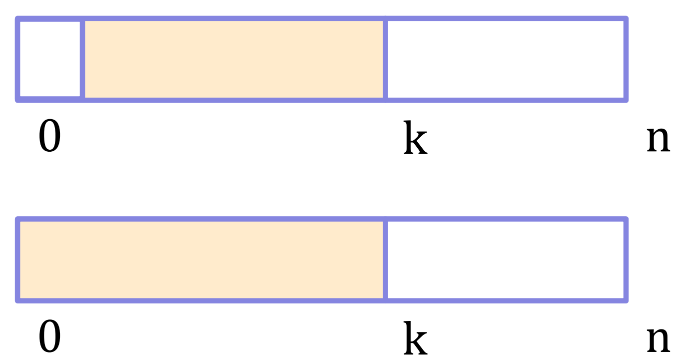
2. **Prove just those "new" facts**

Another Example:

$\{\{\ A[j] < x \text{ for any } 0 < j < k\ \}\}$ **versus**

$\{\{\ A[j] < x \text{ for any } 0 \leq j < k\ \}\}$

– **only need to prove** $A[0] < x$



Finding Loop Invariants

- Loop invariant is often a **weakening** of postcondition…

{{ **Inv**: $s = \text{sum}(A[0 .. j-1])$ … }} — sum of array
{{ **Post**: $s = \text{sum}(A[0 .. n-1])$ }}

{{ **Inv**: $\text{contains}(A[0 .. j-1], x) = F$ … }} — search an array
{{ **Post**: $\text{contains}(A[0 .. n-1], x) = F$ }}

  – but not always…

{{ **Inv**: $A[j] < x$ for any $0 \le j < k$ … }} — search a sorted array
{{ **Post**: $A[j] \ne x$ for any $0 \le j < n$ }}

Array Loop Expectations

In 331, expect you to (eventually) be able to

1. Write invariant that is a simple weakening of postcondition
   - problems of **lower** complexity

2. Write the code, given the idea & invariant
   - problems of **moderate** complexity

3. Check correctness, given code with invariant
   - problems of **higher** complexity
   - (not possible without invariant)

Array Loop Expectations

In 331, expect you to (eventually) be able to

1. Write invariant that is a simple weakening of postcondition
   - problems of lower complexity
   - typical examples:

{{ **Inv**: s = sum(A[0 .. j – 1]) … }} — sum of array
{{ **Post**: s = sum(A[0 .. n – 1]) }}

{{ **Inv**: contains(A[0 .. j – 1], x) = F … }} — search an array
{{ **Post**: contains(A[0 .. n – 1], x) = F }}

Array Loop Expectations

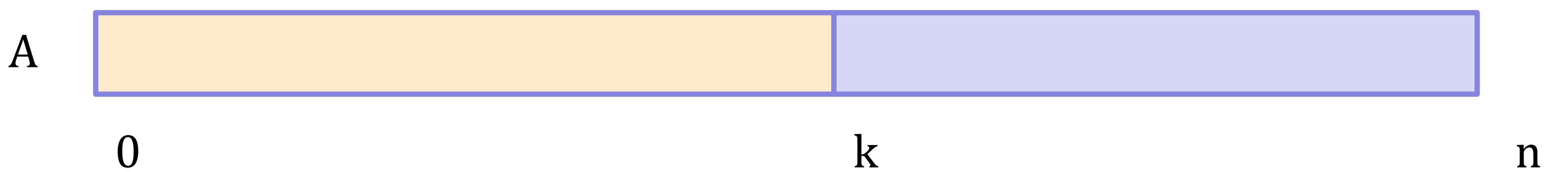
In 331, expect you to (eventually) be able to

1. Write invariant that is a simple weakening of postcondition
   - problems of **lower** complexity

2. Write the code, given the idea & invariant
   - problems of **moderate** complexity

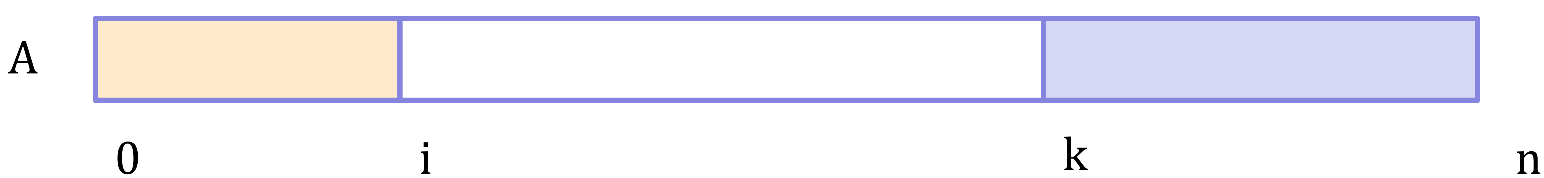
3. Check correctness, given code with invariant
   - problems of **higher** complexity
   - (not possible without invariant)

Searching a Sorted Array (Take Two)



$A[j] < x$ for any $0 \leq j < k$ and $x \leq A[j]$ for any $k \leq j < n$

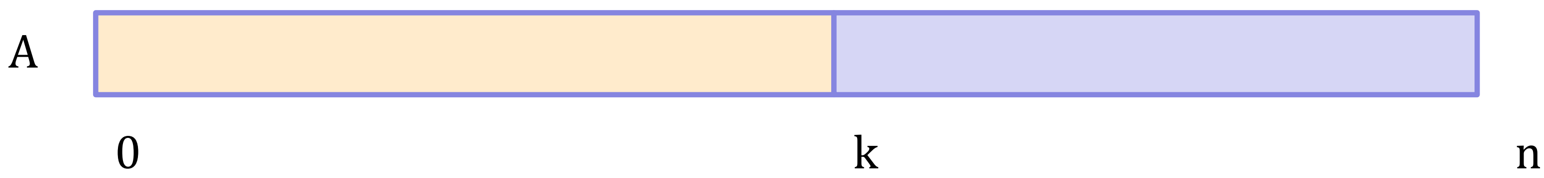
- What is a faster way to search a sorted array?
  - use binary search!
  - invariant looks like this:



$A[j] < x$ for any $0 \leq j < i$

$x \leq A[j]$ for any $k \leq j < n$

Searching a Sorted Array (Take Two)



$A[j] < x$ for any $0 \leq j < k$ and $x \leq A[j]$ for any $k \leq j < n$

- Would not expect you to invent binary search
  - but would expect you can code review an implementation

    all code and the invariant are provided

Array Loop Expectations

In 331, expect you to (eventually) be able to

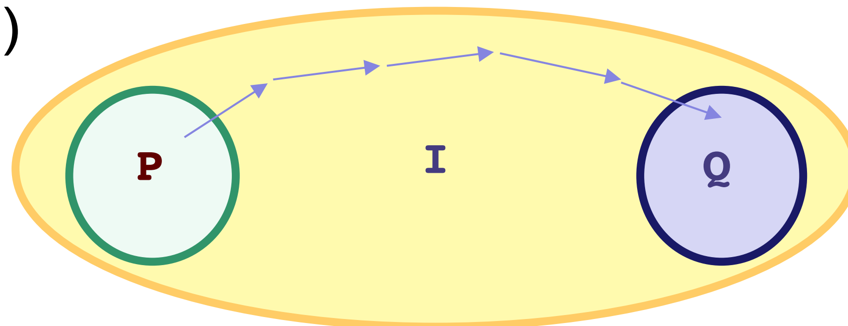
1. Write invariant that is a simple weakening of postcondition
   - problems of **lower** complexity

2. Write the code, given the idea & invariant
   - problems of **moderate** complexity

3. Check correctness, given code with invariant
   - problems of **higher** complexity
   - (not possible without invariant)

From Invariant to Code (Problem Type 2)

- Algorithm Idea formalized in
  - invariant
  - progress step (e.g., $j = j + 1$)



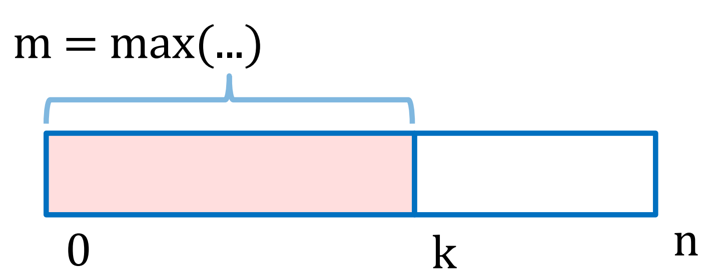
From invariant to code:

1. Write code before loop to make Inv hold initially
2. Write code inside loop to make Inv hold again
3. Choose exit so that "Inv and not cond" implies postcondition

Max of an Array (Problem Type 2)

- **Calculate a number "m" that is the max in array A**

- **Algorithm Idea...**
  - **look through the loop from $k = 0$ up to $n - 1$**
  - **keep track of the maximum of $A[0 .. k-1]$ in "m"**
  - **formalize that in an invariant...**



Max of an Array (Problem Type 2)

- **Calculate a number "m" that is the max in array A**

- **Algorithm Idea...**
  - **look through the loop from $k = 0$ up to $n - 1$**
  - **keep track of the maximum of $A[0 .. k-1]$ in "m"**
  - **m is the maximum of $A[0 .. k-1]$, i.e.,**

$A[j] \leq m$ for any $0 \leq j < k$	m **is at least** $A[0], .., A[k\text{-}1]$
$A[j] = m$ for some $0 \leq j < k$	m **is one of** $A[0], .., A[k\text{-}1]$

- **Invariant references "m" and "k"**
  - **these will be variables in the code**

Max of an Array (Problem Type 2)

{{ **Pre**: n := A.length > 0 }}

```
let k: number = …
let m: number = …
```

What's an easy way to make this hold?

m = A[0] and k = 1

{{ **Inv**: A[j] ≤ m for any 0 ≤ j < k and A[j] = m for **some** 0 ≤ j < k and 0 ≤ k ≤ n }}

```
while (___________) {
  …
  k = k + 1;
}
```

{{ **Post**: A[j] ≤ m for any 0 ≤ j < n and A[j] = m for some 0 ≤ j < n }}

```
return m;
```

Max of an Array (Problem Type 2)

{{ **Pre**: n := A.length > 0 }}

```
let k: number = 1;
let m: number = A[0];
```

{{ **Inv**: A[j] ≤ m for any 0 ≤ j < k and A[j] = m for **some** 0 ≤ j < k and 0 ≤ k ≤ n }}

```
while (___________) {
  …
  k = k + 1;
}
```

What extra fact would make this match Post?

k = n

{{ **Post**: A[j] ≤ m for any 0 ≤ j < n and A[j] = m for some 0 ≤ j < n }}

```
return m;
```

Max of an Array (Problem Type 2)

{{ **Pre**: n := A.length > 0 }}

```
let k: number = 1;
let m: number = A[0];
```

{{ **Inv**: A[j] ≤ m for any 0 ≤ j < k and A[j] = m for **some** 0 ≤ j < k and 0 ≤ k ≤ n }}

```
while (k < n) {
  ...
  k = k + 1;
}
```

{{ **Post**: A[j] ≤ m for any 0 ≤ j < n and A[j] = m for some 0 ≤ j < n }}

```
return m;
```

Max of an Array (Problem Type 2)

{{ **Pre**: n := A.length > 0 }}

```
let k: number = 1;
let m: number = A[0];
```

{{ **Inv**: A[j] ≤ m for any 0 ≤ j < k and A[j] = m for **some** 0 ≤ j < k and 0 ≤ k ≤ n }}

```
while (k < n) {
```

{{ A[j] ≤ m for any 0 ≤ j < k and A[j] = m for some 0 ≤ j < k and 0 ≤ k < n }}

…

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

{{ A[j] ≤ m for any 0 ≤ j < k and A[j] = m for some 0 ≤ j < k and 0 ≤ k ≤ n }}

```
}
```

{{ **Post**: A[j] ≤ m for any 0 ≤ j < n and A[j] = m for some 0 ≤ j < n }}

```
return m;
```

Max of an Array (Problem Type 2)

{{ **Pre**: n := A.length > 0 }}

```
let k: number = 1;
let m: number = A[0];
```

{{ **Inv**: A[j] ≤ m for any 0 ≤ j < k and A[j] = m for some 0 ≤ j < k and 0 ≤ k ≤ n }}

```
while (k < n) {
```

{{ A[j] ≤ m for any 0 ≤ j < k and A[j] = m for some 0 ≤ j < k and 0 ≤ k < n }}

…

{{ A[j] ≤ m for any 0 ≤ j < k+1 and A[j] = m for some 0 ≤ j < k+1 and 0 ≤ k+1 ≤ n }}

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

{{ A[j] ≤ m for any 0 ≤ j < k and A[j] = m for some 0 ≤ j < k and 0 ≤ k ≤ n }}

```
}
```

{{ **Post**: A[j] ≤ m for any 0 ≤ j < n and A[j] = m for some 0 ≤ j < n }}

```
return m;
```

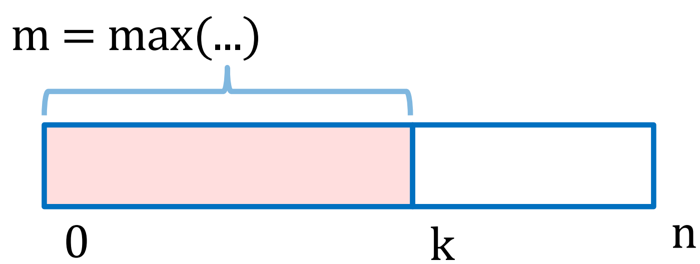
Max of an Array (Problem Type 2)

{{ $A[j] \leq m$ for any $0 \leq j < k$ and $A[j] = m$ for some $0 \leq j < k$ and $0 \leq k < n$ }}

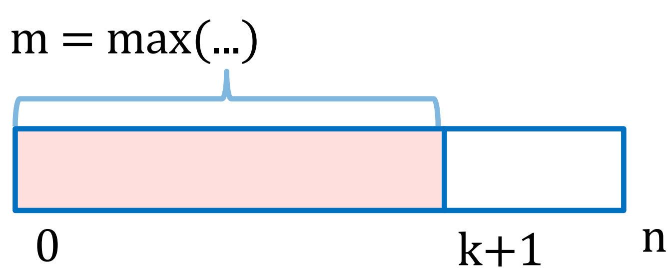
…

{{ $A[j] \leq m$ for any $0 \leq j < k+1$ and $A[j] = m$ for some $0 \leq j < k+1$ and $0 \leq k+1 \leq n$ }}

Top:



Bottom:



Tricky because max(..) involves two sets of facts (one "for any" and one "for some")

Max of an Array (Problem Type 2)

$\{\{$ A[j] $\le$ m for any $0 \le j < k$ and A[j] = m for some $0 \le j < k$ and $0 \le k < n$ $\}\}$

…

$\{\{$ A[j] $\le$ m for any $0 \le j < k+1$ and A[j] = m for some $0 \le j < k+1$ and $0 \le k+1 \le n$ $\}\}$

Step 1: What facts are new in the bottom assertion?

Just $A[k] \le m$

Note that second part is weakened

from $A[j] = m$ for some $0 \le j < k$

to $A[j] = m$ for some $0 \le j < k+1$

Now, we can have $A[k] = m$, when we couldn't before.

What code do we write to ensure $A[k] \le m$?

Max of an Array (Problem Type 2)

```
while (k != n) {
  {{ A[j] ≤ m for any 0 ≤ j < k and A[j] = m for some 0 ≤ j < k and 0 ≤ k < n }}
  if (A[k] > m)
    m = A[k];
  {{ A[j] ≤ m for any 0 ≤ j < k+1 and A[j] = m for some 0 ≤ j < k+1 and 0 ≤ k+1 ≤ n }}
  k = k + 1;
}
```

Step 1: What facts are new in the bottom assertion?

Just $A[k] \leq m$

Else branch happens if $A[k] \leq m$

Then branch makes that true by setting $m = A[k]$
Still have $A[j] = m$ for some j, namely, $j = k$

Max of an Array (Problem Type 2)

{{ **Pre**: n := A.length > 0 }}

```
let k: number = 0;
let m: number = A[0];
```

{{ **Inv**: A[j] ≤ m for any 0 ≤ j < k and A[j] = m for some 0 ≤ j < k and 0 ≤ k ≤ n }}

```
while (k < n) {
  if (A[k] > m)
    m = A[k];
  k = k + 1;
}
```

{{ **Post**: A[j] ≤ m for any 0 ≤ j < n and A[j] = m for some 0 ≤ j < n }}

```
return m;
```