CSE 331

Arrays

Katherine Murphy

Credits: Profs. Kevin Zatloukal and James Wilcox

Recall: Turning Recursion Into a Loop

- Saw templates for structural recursion on
  - natural numbers — straightforward
  - lists — harder

- Special case for tail recursion on
  - lists — straightforward

Processing Lists with Loops

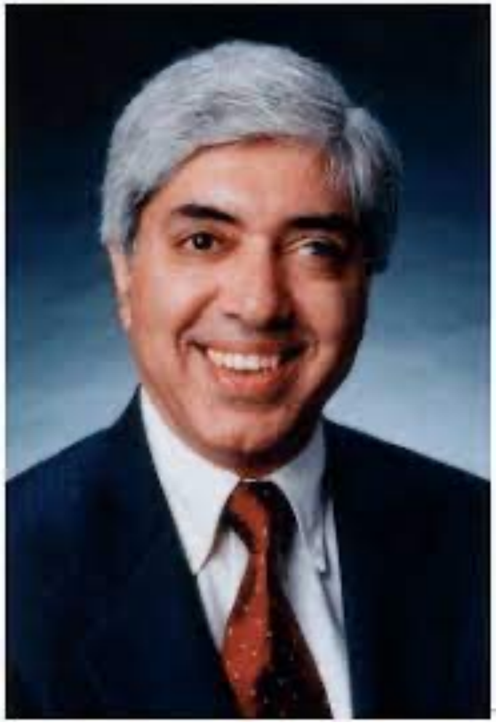
- **Hard to process lists with loops**
  - **only have easy access to the last element added**

    natural processing would start from the other end
  - **must reverse the list to work “bottom up”**

    that requires an additional O(n) space

- **There is an easier way to fix this…**
  - **switch data structures**
  - **use one that lets us access either end easily**

"Lists are the original data structure for functional programming,
just as arrays are the original data structure of imperative programming"



Ravi Sethi

Array Accesses

- **Easily access both $A[0]$ and $A[n\text{-}1]$, where $n = A.length$**
  - **bottom-up loops are now easy**

- **"With great power, comes great responsibility"**

  — the Peter Parker Principle

- **Whenever we write "$A[j]$", we must check $0 \leq j < n$**
  - **new bug just dropped!**

    with list, we only need to worry about nil and non-nil

    once we know L is non-nil, we know L.hd exists
  - **TypeScript will not help us with this!**

    type checker does catch "could be nil" bugs, but not this

Correctness Levels

Level	Description	Testing	Tools	Reasoning
0	small # of inputs	exhaustive		
1	straight from spec	heuristics	type checking	code reviews
2	no mutation	“	libraries	calculation induction
3	local variable mutation	“	“	Floyd logic
4	array mutation	“	“	for-any facts
5	heap state mutation	“	“	rep invariants

Array Math

- Easily access both $\mathrm{A}[0]$ and $\mathrm{A}[\mathrm{n}\text{-}1]$, where $\mathrm{n} = \mathrm{A.length}$
  - bottom-up loops are now easy

- "With great power, comes great responsibility"

  — the Peter Parker Principle

- Will need new tools for reasoning about arrays
  - will start with new math for describing them

Array Literals

- **Write array values in math like this:**

$$A := [1, 2, 3] \qquad\qquad (\text{with } A : \text{Array}_{\mathbb{Z}})$$

  - **the empty array is "[ ]"**

- **Array literal syntax is the same in TypeScript:**

```
const A: Array<bigint> = [1n, 2n, 3n];
const B: bigint[] = [4n, 5n];
```

  - **can write** $\text{Array}_{\mathbb{Z}}$ **as "**`Array<bigint>`**" or "**`bigint[]`**"**

Array Concatenation

- Define the operation "$+\!\!+$" as array concatenation
  - makes clear the arguments are arrays, not numbers

- The following properties hold for any arrays A, B, C

$$\mathrm{A} +\!\!+ [] = \mathrm{A} = [] +\!\!+ \mathrm{A} \qquad \text{("identity")}$$

$$\mathrm{A} +\!\!+ (\mathrm{B} +\!\!+ \mathrm{C}) = (\mathrm{A} +\!\!+ \mathrm{B}) +\!\!+ \mathrm{C} \qquad \text{("associativity")}$$

  - we will use these facts *without* explanation in calculations
  - second line says parentheses *don't matter*, so we will write $\mathrm{A} +\!\!+ \mathrm{B} +\!\!+ \mathrm{C}$ and not say where the $(..)$ go

Array Concatenation Math

- **Same properties hold for lists**

$[] \mathbin{+\!\!+} A = A$ | $\text{concat}(\text{nil}, L) = L$

$A \mathbin{+\!\!+} [] = A$ | $\text{concat}(L, \text{nil}) = L$

$A \mathbin{+\!\!+} (B \mathbin{+\!\!+} C) = (A \mathbin{+\!\!+} B) \mathbin{+\!\!+} C$ | $\text{concat}(A, \text{concat}(B, C))$
$= \text{concat}(\text{concat}(A, B), C)$

  - **we required explanation of these facts for lists**
  - **but we will <u>not</u> require explanation of these facts for arrays**

    (trying to reason more quickly, now that we have more practice)

Defining Functions on Arrays

- Can still define functions recursively

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

  – could write patterns with "$[y] \mathbin{+\!\!+} A$" instead

Subarrays

- **Often useful to talk about part of an array (subarray)**
  - **define the following notation**

    $A[i .. j] = [\ A[i], A[i+1], \ldots, A[j]\ ]$

  - **note that this includes** $A[j]$

    (some functions exclude the right end; we will include it)

Subarrays

$A[i \mathinner{..} j] = [\ A[i], A[i+1], \ldots, A[j]\ ]$

- **Define this formally as follows**

$$
\begin{aligned}
&\textbf{func } A[i \mathinner{..} j] && := [] && \textbf{if } j < i \\
&\phantom{\textbf{func }} A[i \mathinner{..} j] && := A[i \mathinner{..} j\text{-}1] \mathbin{+\!\!+} [A[j]] && \textbf{if } i \leq j
\end{aligned}
$$

  - **second case needs $0 \leq j < n$ for this to make sense**

    $A[i \mathinner{..} j]$ is undefined if $i \leq j$ and $(i < 0 \text{ or } n \leq j)$

  - **note that $A[0 \mathinner{..} \text{-}1] = []$ since $\text{-}1 < 0$**

    "Isn't -1 an array out of bounds error?"

    In code, yes — In math, no (the definition says this is an empty array)

Subarray Math

$$
\begin{array}{lll}
\textbf{func}\ A[i\,..\,j] & := [] & \textbf{if}\ j < i \\
A[i\,..\,j] & := A[i\,..\,j\text{-}1] \mathbin{+\!\!\!+} [A[j]] & \textbf{if}\ 0 \le i \le j < A.\text{length} \\
A[i\,..\,j] & := \text{undefined} & \textbf{if}\ i \le j \text{ and } (i < 0 \text{ or } A.\text{length} \le j)
\end{array}
$$

- **Some useful facts**

$$A = A[0\,..\,n\text{-}1] \qquad (= [A[0], A[1], \ldots, A[n\text{-}1]])$$
$$\text{where } n = A.\text{length}$$

  - **the subarray from 0 to $n - 1$ is the entire array**

$$A[i\,..\,j] = A[i\,..\,k] \mathbin{+\!\!\!+} A[k+1\,..\,j]$$

  - **holds for any $i, j, k : \mathbb{N}$ satisfying $i - 1 \le k \le j$ (and $0 \le i \le j < n$)**
  - **we will use these *without* explanation**

TypeScript Arrays

- **Translating math to TypeScript**

Math	**TypeScript**
$A \mathbin{+\!\!\!+} B$	`A.concat(B)`
$A[i \mathinner{..} j]$	`A.slice(i, j+1)`

  - **JavaScript's `A.slice(i, j)` does not include $A[j]$, so we need to increase `j` by one**

- **Note: array out of bounds does not throw Error**
  - **returns `undefined`**

    (hope you like debugging!)

Facts About Arrays

- **"With great power, comes great responsibility"**

- **Since we can easily access any $\mathrm{A}[\mathrm{j}]$,**
  may need to keep track of facts about it
  - **may need facts about *every* element in the array**

    applies to preconditions, postconditions, and intermediate assertions

- **We can write facts about several elements at once:**
  - **this says that elements at indexes 2 .. 10 are non-negative**

    $$0 \leq \mathrm{A}[\mathrm{j}] \text{ for any } 2 \leq \mathrm{j} \leq 10$$

  - **shorthand for 9 facts** $(0 \leq \mathrm{A}[2], \ldots, 0 \leq \mathrm{A}[10])$

Finding an Element in an Array

- **Can search for an element in an array as follows**

func contains([], x)	:= F		**for any ...**
contains(A ⧺ [y], x)	:= T	**if** $x = y$	**for any ...**
contains(A ⧺ [y], x)	:= contains(A, x)	**if** $x \neq y$	**for any ...**

- **Searches through the array in linear time**
  - **did the same on lists**

- **Can search more quickly if the list is sorted**
  - **precondition is** $A[0] \leq A[1] \leq \ldots \leq A[n\text{-}1]$ **(informal)**
  - **write this formally as**

$$A[j] \leq A[j+1] \text{ for any } 0 \leq j \leq n - 2$$

Loops with Arrays

Sum of an Array

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

- **Could translate this directly into a recursive function**
  - **that would be straight from the spec**

- **Do this instead with a loop. Loop idea…**
  - **use the “bottom up” approach**
  - **start from $[]$ and work up to all of A**
  - **at any point, we have $\mathrm{sum}(A[0 .. j\text{-}1])$ for some index j**

    I will add one extra fact we also need

Sum of an Array

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

- **Loop implementation:**

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

could write "`j !== A.length`"
but this is normal

Sum of an Array

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

- **Loop implementation:**

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

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

$$\begin{aligned} s &= 0 \\ &= \mathrm{sum}([]) && \textbf{def of } \mathrm{sum} \\ &= \mathrm{sum}(A[0 \mathinner{..} -1]) \\ &= \mathrm{sum}(A[0 \mathinner{..} j-1]) && \textbf{since } j = 0 \end{aligned}$$

$$\begin{aligned} j &= 0 \\ &\leq A.\mathrm{length} \end{aligned}$$

Sum of an Array

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

- **Loop implementation:**

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

Sum of an Array

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

- **Loop implementation:**

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

$$\begin{aligned} s &= \mathrm{sum}(A[0 \mathinner{..} j - 1]) \\ &= \mathrm{sum}(A[0 \mathinner{..} A.\mathrm{length} - 1]) \\ &= \mathrm{sum}(A) \end{aligned}$$

Sum of an Array

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

- **Loop implementation:**

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

Sum of an Array

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

- **Loop implementation:**

```
while (j < A.length) {
  {{ s = sum(A[0 .. j – 1]) and 0 ≤ j < A.length }}
  s = s + A[j];
  {{ s – A[j] = sum(A[0 .. j – 1]) and 0 ≤ j < A.length }}
  j = j + 1n;
  {{ s = sum(A[0 .. j – 1]) and 0 ≤ j ≤ A.length }}
}
```

Sum of an Array

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

- **Loop implementation:**

```
while (j < A.length) {
  {{ s = sum(A[0 .. j – 1]) and 0 ≤ j < A.length }}
  s = s + A[j];
  {{ s – A[j] = sum(A[0 .. j – 1]) and 0 ≤ j < A.length }}
  j = j + 1n;
  {{ s – A[j – 1] = sum(A[0 .. j – 2]) and 0 ≤ j – 1 < A.length }}
  {{ s = sum(A[0 .. j – 1]) and 0 ≤ j ≤ A.length }}
}
```

Sum of an Array

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

- **Loop implementation:**

```
while (j < A.length) {
  {{ s = sum(A[0 .. j – 1]) and 0 ≤ j < A.length }}
  s = s + A[j];
  {{ s – A[j] = sum(A[0 .. j – 1]) and 0 ≤ j < A.length }}
  j = j + 1n;
  {{ s – A[j – 1] = sum(A[0 .. j – 2]) and 0 ≤ j – 1 < A.length }}
  {{ s = sum(A[0 .. j – 1]) and 0 ≤ j ≤ A.length }}
}
```

$$
\begin{aligned}
s &= \mathrm{sum}(A[0 \mathinner{..} j-2]) + A[j-1] && \textbf{since } s - A[j\text{-}1] = \mathrm{sum}(..) \\
&= \mathrm{sum}(A[0 \mathinner{..} j-2] \mathbin{+\!\!+} [A[j-1]]) && \textbf{def of } \mathrm{sum} \\
&= \mathrm{sum}(A[0 \mathinner{..} j-1])
\end{aligned}
$$

Recursion versus Loops

- There is a fundamental tension between:
  - Natural recursive order (bottom-up, aka back-to-front)
  - Natural loop order (front-to-back)

- Three ways to bridge this gap:
  - Make the loop serve the recursion
    Bottom-up list loop template calling $\mathrm{rev}(\mathrm{L})$
  - Make the recursion serve the loop
    Tail recursion
  - Change the data structure
    Arrays

Recursion versus Loops

- Three ways to bridge this gap:
  - Make the loop serve the recursion

$$
\begin{array}{lll}
\textbf{func}\ \mathrm{sum}(\mathrm{nil}) & := 0 & \\
\quad\ \mathrm{sum}(\mathrm{cons}(x, L)) & := x + \mathrm{sum}(L) & \text{for any } x : \mathbb{Z} \text{ and } L : \mathrm{List}
\end{array}
$$

  - Make the recursion serve the loop

$$
\begin{array}{lll}
\textbf{func}\ \text{sum-acc}(\mathrm{nil}, s) & := s & \\
\quad\ \text{sum-acc}(\mathrm{cons}(x, L), s) & := \text{sum-acc}(L, x + s) & \text{for any } x : \mathbb{Z} \text{ and } L : \mathrm{List}
\end{array}
$$

  - Change the data structure

$$
\begin{array}{lll}
\textbf{func}\ \mathrm{sum}([]) & := 0 & \\
\quad\ \mathrm{sum}(A \mathbin{+\!\!+} [y]) & := \mathrm{sum}(A) + y & \text{for any } y : \mathbb{Z} \text{ and } A : \mathrm{Array}_{\mathbb{Z}}
\end{array}
$$

Linear Search of an Array

$$
\begin{array}{llll}
\textbf{func} \ \text{contains}([], x) & := F & \\
\quad \text{contains}(A \mathbin{+\!\!\!+} [y], x) & := T & \text{if } x = y \\
\quad \text{contains}(A \mathbin{+\!\!\!+} [y], x) & := \text{contains}(A, x) & \text{if } x \neq y
\end{array}
$$

- **Could translate this directly into a recursive function**
  - **that would be straight from the spec**

- **Do this instead with a loop. Loop idea...**
  - **use the "bottom up" template**
  - **start from** $[]$ **and work up to all of** A
  - **but we can stop immediately if we find x**

    contains returns true in that case
  - **otherwise, we have** $\text{contains}(A[0 \,..\, j{-}1], x) = F$ **for some j**

Linear Search of an Array

$$
\begin{array}{llll}
\textbf{func}\ \text{contains}([], x) & := \text{F} & \\
\quad \text{contains}(A + [y], x) & := \text{T} & \text{if } x = y \\
\quad \text{contains}(A + [y], x) & := \text{contains}(A, x) & \text{if } x \neq y
\end{array}
$$

- **Loop implementation:**

```
let j: bigint = 0n;
{{ 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 + 1n;
}
{{ contains(A, x) = F }}
return false;
```

Linear Search of an Array

$$
\begin{array}{lll}
\textbf{func}\ \text{contains}([\,], x) & := F & \\
\quad \text{contains}(A \mathbin{+\!\!+} [y], x) & := T & \text{if } x = y \\
\quad \text{contains}(A \mathbin{+\!\!+} [y], x) & := \text{contains}(A, x) & \text{if } x \neq y
\end{array}
$$

- **Loop implementation:**

```
let j: bigint = 0n;
{{ j = 0 }}
{{ Inv: contains(A[0 .. j–1], x) = F and 0 ≤ j ≤ A.length }}
while (j < A.length) {
  if (A[j] === x)
    return true;
  j = j + 1n;
}
return false;
```

Linear Search of an Array

$$
\begin{array}{lll}
\textbf{func } \text{contains}([], x) & := F & \\
\quad \text{contains}(A \mathbin{+\!\!+} [y], x) & := T & \text{if } x = y \\
\quad \text{contains}(A \mathbin{+\!\!+} [y], x) & := \text{contains}(A, x) & \text{if } x \neq y
\end{array}
$$

- **Loop implementation:**

```
let j: bigint = 0n;
{{ j = 0 }}
{{ Inv: contains(A[0 .. j–1], x) = F and 0 ≤ j ≤ A.length }}
while (j < A.length) {
  if (A[j] === x)
    return true;
  j = j + 1n;
}
return false;
```

$$
\begin{array}{ll}
\text{contains}(A[0 \,..\, j-1], x) & \\
\quad = \text{contains}(A[0 \,..\, -1], x) & \textbf{since } j = 0 \\
\quad = \text{contains}([], x) & \\
\quad = F & \textbf{def of } \text{contains}
\end{array}
$$

$0 \leq 0 = j$ and $j = 0 \leq A.\text{length}$

Linear Search of an Array

$$
\begin{array}{llll}
\textbf{func}\ \text{contains}([], x) & := F & \\
\quad \text{contains}(A +\!\!\!+ [y], x) & := T & \text{if } x = y \\
\quad \text{contains}(A +\!\!\!+ [y], x) & := \text{contains}(A, x) & \text{if } x \neq y
\end{array}
$$

- **Loop implementation:**

```
let j: bigint = 0n;
{{ Inv: contains(A[0 .. j–1], x) = F and 0 ≤ j ≤ A.length }}
while (j < A.length) {
  if (A[j] === x)
    return true;
  j = j + 1n;
}
{{ contains(A[0 .. j–1], x) = F and j = A.length }}
{{ contains(A, x) = F }}
return false;
```

Linear Search of an Array

$$
\begin{array}{lll}
\textbf{func } \text{contains}([], x) & := F & \\
\quad \text{contains}(A + [y], x) & := T & \text{if } x = y \\
\quad \text{contains}(A + [y], x) & := \text{contains}(A, x) & \text{if } x \neq y
\end{array}
$$

- **Loop implementation:**

```
let j: bigint = 0n;
{{ Inv: contains(A[0 .. j–1], x) = F and 0 ≤ j ≤ A.length }}
while (j < A.length) {
  if (A[j] === x)
    return true;
  j = j + 1n;
}
{{ contains(A[0 .. j–1], x) = F and j = A.length }}
{{ contains(A, x) = F }}
return false;
```

$F = \text{contains}(A[0 .. j-1], x)$
$= \text{contains}(A[0 .. A.\text{length} - 1], x)$ since $j = \ldots$
$= \text{contains}(A, x)$

Linear Search of an Array

$$\begin{array}{llll} \textbf{func}\ \mathrm{contains}([\,], x) & := \mathrm{F} & \\ \quad \mathrm{contains}(A \mathbin{+\!\!+} [y], x) & := \mathrm{T} & \text{if } x = y \\ \quad \mathrm{contains}(A \mathbin{+\!\!+} [y], x) & := \mathrm{contains}(A, x) & \text{if } x \neq y \end{array}$$

- **Loop implementation:**

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

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 $x \neq y$

- **Loop implementation:**

{{ contains(A[0 .. j−1], x) = F and $0 \leq j < A.\text{length}$ }}

```
if (A[j] === x) {
```

{{ contains(A, x) = T }}

```
  return true;
} else {
}
j = j + 1n;
```

{{ contains(A[0 .. j−1], x) = F and $0 \leq j \leq A.\text{length}$ }}

Linear Search of an Array

$$
\begin{array}{lll}
\textbf{func}\ \mathrm{contains}([\,], x) & := \mathrm{F} & \\
\quad \mathrm{contains}(A \mathbin{+\!\!\!+} [y], x) & := \mathrm{T} & \text{if } x = y \\
\quad \mathrm{contains}(A \mathbin{+\!\!\!+} [y], x) & := \mathrm{contains}(A, x) & \text{if } x \neq y
\end{array}
$$

- **Loop implementation:**

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

```
if (A[j] === x) {
```

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

{{ contains(A, x) = T }}

```
  return true;
} else {
```

…

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:**

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

```
if (A[j] === x) {
```

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

{{ contains(A, x) = T }}

```
  return true;
} else {
```

...

contains(A[0 .. j], x)
= contains(A[0 .. j-1] ⧺ [A[j]], x)
= T   **since** A[j] = x

Can now prove by **induction** that contains(A, x) = T

Linear Search of an Array

$$
\begin{array}{llll}
\textbf{func}\ \text{contains}([], x) & := F & \\
\quad \text{contains}(A +\!\!+ [y], x) & := T & \text{if } x = y \\
\quad \text{contains}(A +\!\!+ [y], x) & := \text{contains}(A, x) & \text{if } x \neq y
\end{array}
$$

- **Loop implementation:**

{{ contains(A[0 .. j–1], x) = F and j < A.length }}

```
if (A[j] === x) {
  return true;
} else {
```

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

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

```
}
```

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

```
j = j + 1;
```

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

Linear Search of an Array

$$
\begin{array}{llll}
\textbf{func}\ \text{contains}([], x) & := F & \\
\quad \text{contains}(A + [y], x) & := T & \text{if } x = y \\
\quad \text{contains}(A + [y], x) & := \text{contains}(A, x) & \text{if } x \neq y
\end{array}
$$

- **Loop implementation:**

{{ contains(A[0 .. j–1], x) = F and j < A.length }}

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

Linear Search of an Array

$$\begin{aligned}
&\textbf{func}\ \text{contains}([], x) && := F && \\
&\quad\ \text{contains}(A \mathbin{+\!\!+} [y], x) && := T && \text{if } x = y \\
&\quad\ \text{contains}(A \mathbin{+\!\!+} [y], x) && := \text{contains}(A, x) && \text{if } x \neq y
\end{aligned}$$

- **Loop implementation:**

{{ contains(A[0 .. j–1], x) = F and j < A.length }}

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

$$\begin{aligned}
F &= \text{contains}(A[0 .. j-1], x) \\
&= \text{contains}(A[0 .. j-1] \mathbin{+\!\!+} [A[j]], x) && \textbf{def of } \text{contains } (\textbf{since } A[j] \neq x) \\
&= \text{contains}(A[0 .. j], x)
\end{aligned}$$

Loop Invariants with Arrays

- **Saw two more examples previously**

  {{ **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**: $\text{contains}(A[0 .. n-1], x) = F$ }}

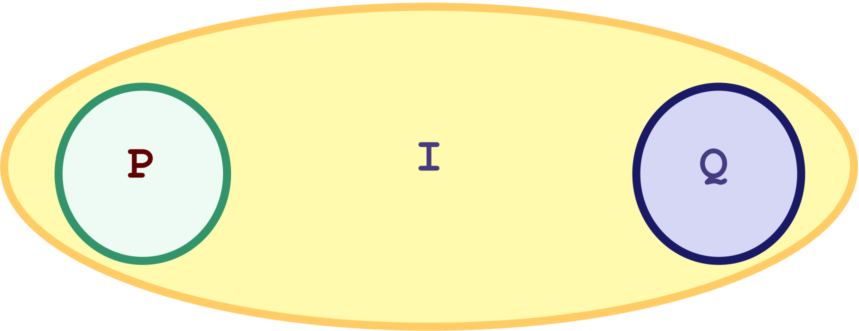
  - **in both cases,** Post **is a special case of** Inv **(where** $j = n$**)**
  - **in other words,** Inv **is a weakening of** Post

- **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 }}
```



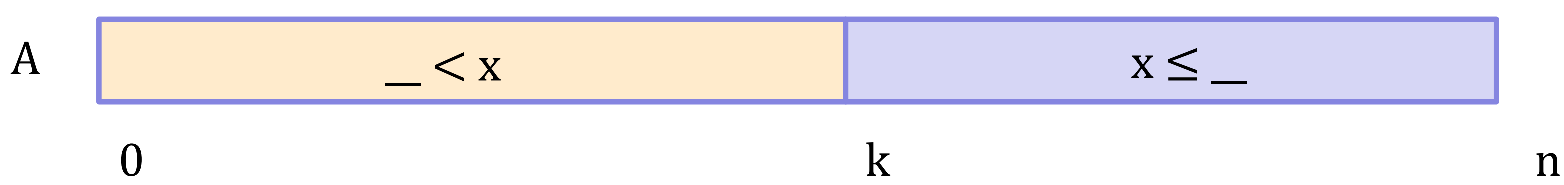
  - usually are many ways to weaken it…

Searching a Sorted Array

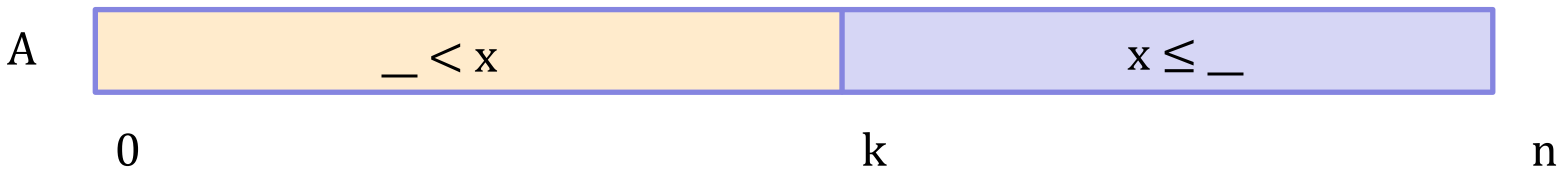
- **Suppose we require A to be sorted:**
  - **precondition includes**

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

- **Want to find the index k where "x" would be...**
  - **picture would look like this:**



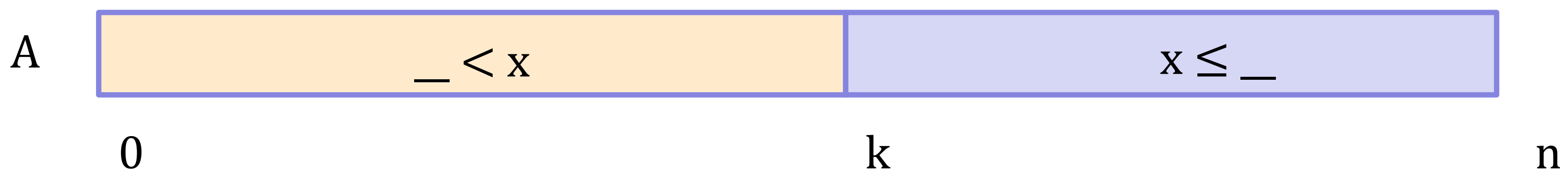
Aside: Array Facts as Pictures



- **Can use pictures to write array facts *concisely***
  - **one thing that whiteboard in your office is good for**

- **Example above encodes several facts:**
  - $A[j] < x$ for any $0 \leq j < k$
  - $x \leq A[j]$ for any $k \leq j < n$
  - $0 \leq k \leq n$

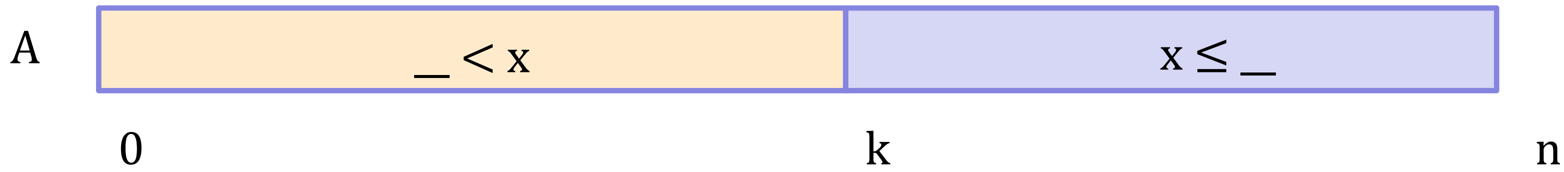
Searching a Sorted Array



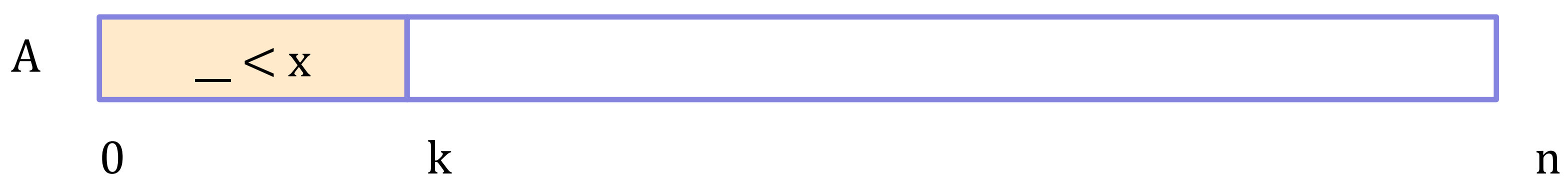
- **End with complete knowledge of** $\mathrm{A[j]}$ **vs** x
  - **how can we describe *partial* knowledge?**

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

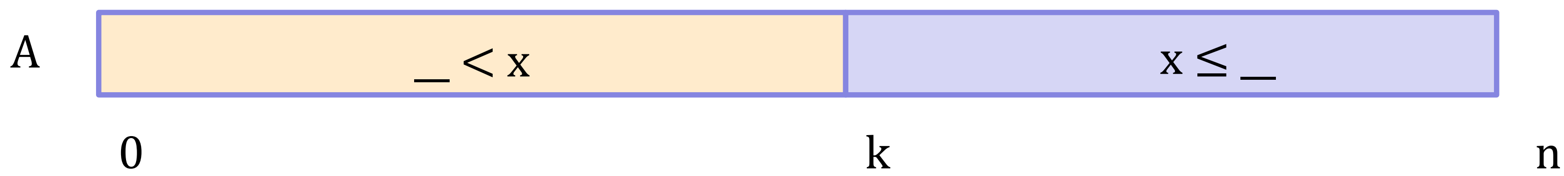
Searching a Sorted Array



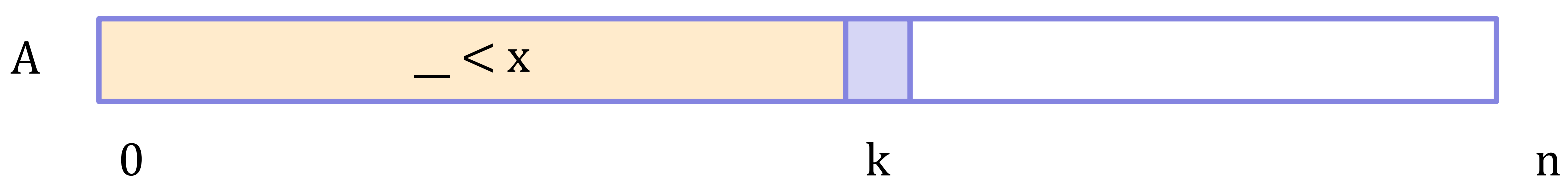
- End with complete knowledge of $A[j]$ vs x
  - how can we describe *partial* knowledge?
  - we will focus on the elements that are smaller than x



Searching a Sorted Array



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



- **Loop idea... increase k until we hit $x \leq A[k]$**

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: bigint = 0n;
{{ 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 + 1n;
  }
}
return false;
```

Searching a Sorted Array

```
let k: bigint = 0n;
{{ 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 + 1n;
  }
}
return false;
```

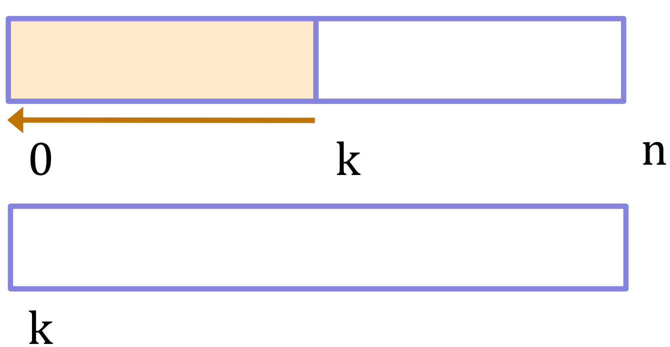
What is the claim when $k = 0$?

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

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

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: bigint = 0n;
{{ 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 + 1n;
  }
}
{{ 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 + 1n;
  }
}
```

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

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

```
return false;
```

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

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

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

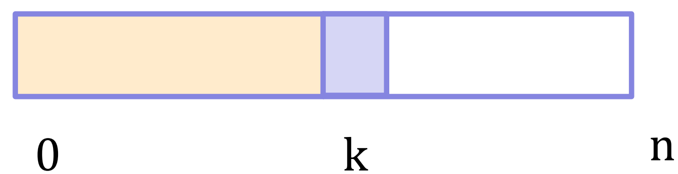
Searching a Sorted Array

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

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

{{ $A[j] \neq x$ 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 $A[k] \le A[k+1] \le \ldots$

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

This means $A[j] \neq x$ 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 + 1n;
  }
}
```

$\{\{$ 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;
```

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

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

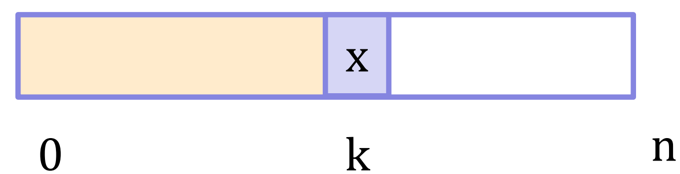
holds in general.

Searching a Sorted Array

```
let k: bigint = 0n;
{{ 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 + 1n;
  }
  {{ 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;
  }
```

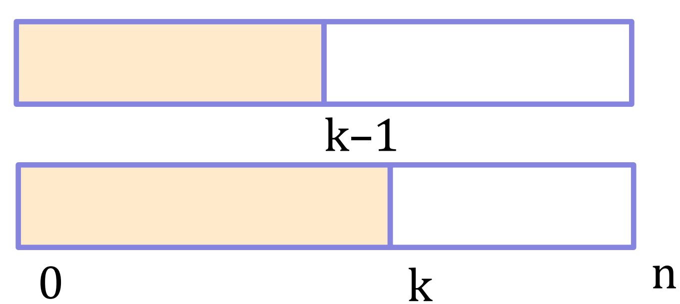


Is the postcondition true?

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 + 1n;
    {{ 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 + 1n;
    {{ 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

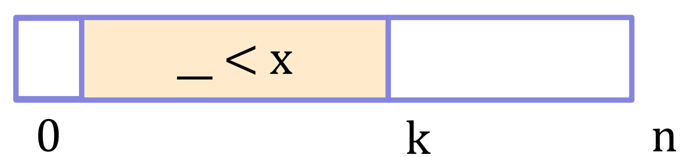
2. Prove just those “new” facts

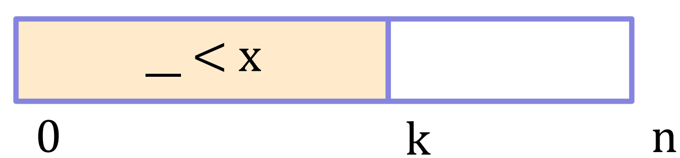
Another Example:

$\{\{\ A[j] < x \text{ for any } 0 < j < k\ \}\}$ **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**: $s = \text{sum}(A[0 .. j-1])$ … }}
{{ **Post**: $s = \text{sum}(A[0 .. n-1])$ }}

sum of array

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

search an array

  - but not always…

{{ **Inv**: $A[j] < x$ for any $0 \leq j < k$ … }}
{{ **Post**: $A[j] \neq x$ for any $0 \leq j < n$ }}

search a sorted array

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

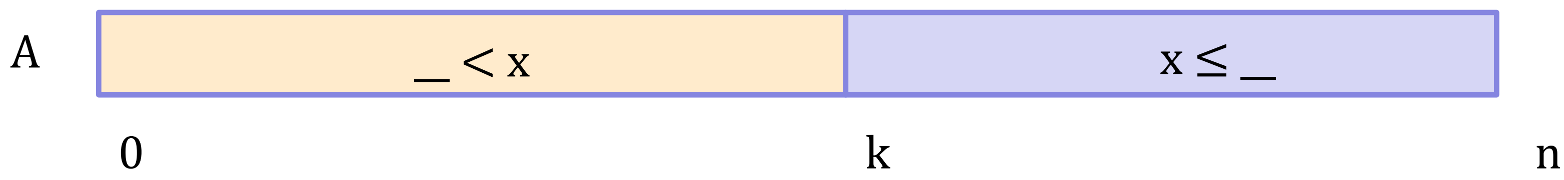
In 331, expect you to (eventually) be able to

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

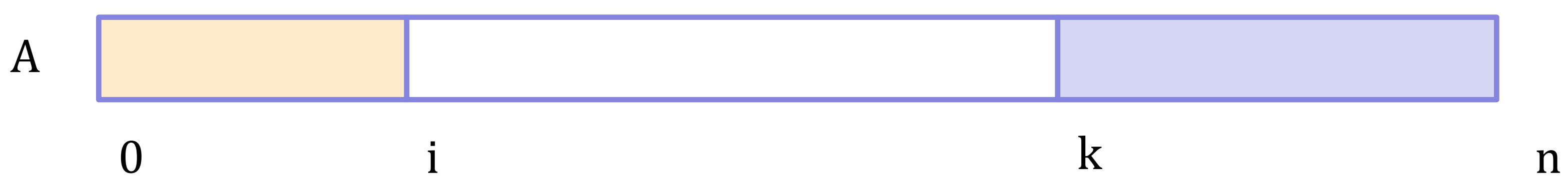
2. Write the code, given the idea & invariant
   - problems of **moderate** complexity

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

Searching a Sorted Array (Take Two)



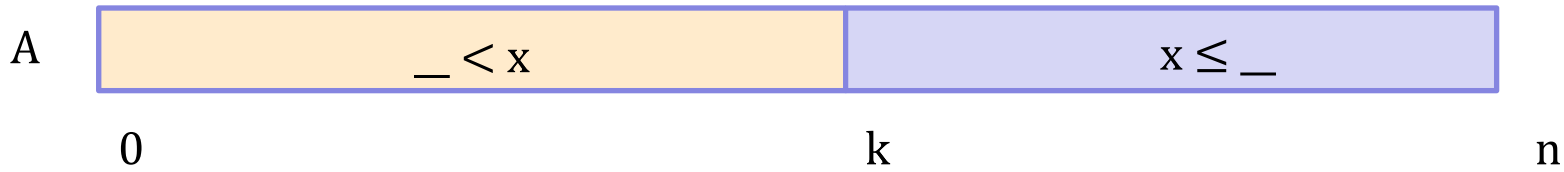
- **What is a faster way to search a sorted array?**
  - **use binary search!**
  - **invariant looks like this:**



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

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

Searching a Sorted Array (Take Two)



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

    all code and the invariant are provided

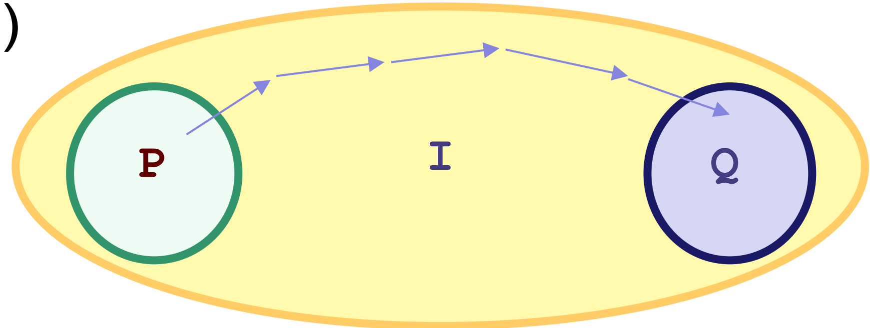
Array Loop Expectations

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., $j = j + 1$)

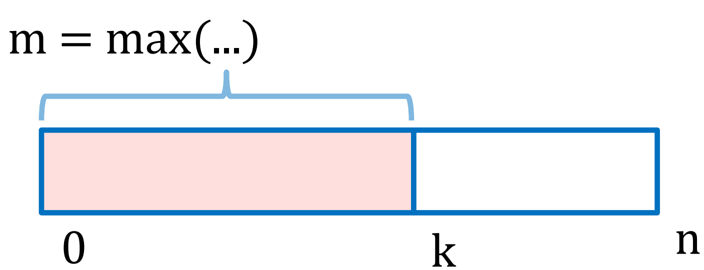


From invariant to code:

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** $A[0], .., A[k-1]$
$A[i] = m$ and $0 \leq i < k$	m **is some** $A[i]$ **in this range**

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

Max of an Array (Problem Type 2)

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

```
let k: bigint = …
let i: bigint = …
let m: bigint = …
```

What's an easy way to make this hold?

k = 1 and i = 0 and m = A[i]

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

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

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

```
return m;
```

Max of an Array (Problem Type 2)

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

```
let k: bigint = 1n;
let i: bigint = 0n;
let m: bigint = A[0];
```

{{ **Inv**: ($A[j] \leq m$ for any $0 \leq j < k$) and $A[i] = m$ and $0 \leq i < k \leq n$ }}

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

What extra fact would make Inv match Post?

$k = n$

{{ **Post**: ($A[j] \leq m$ for any $0 \leq j < n$) and $A[i] = m$ and $0 \leq i < n$ }}

```
return m;
```

Max of an Array (Problem Type 2)

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

```
let k: bigint = 1n;
let i: bigint = 0n;
let m: bigint = A[0];
```

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

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

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

```
return m;
```

Max of an Array (Problem Type 2)

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

```
let k: bigint = 1n;
let i: bigint = 0n;
let m: bigint = A[i];
```

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

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

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

…

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

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

```
}
```

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

```
return m;
```

Max of an Array (Problem Type 2)

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

```
let k: bigint = 1n;
let i: bigint = 0n;
let m: bigint = A[0];
```

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

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

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

…

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

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

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

```
}
```

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

```
return m;
```

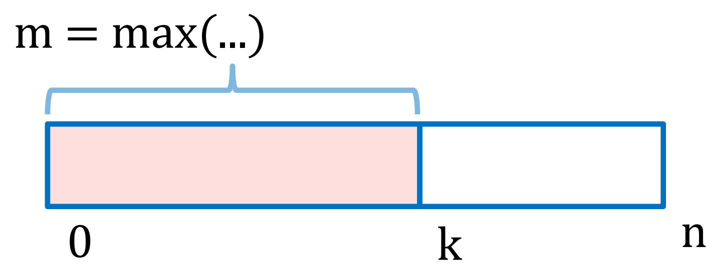
Max of an Array (Problem Type 2)

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

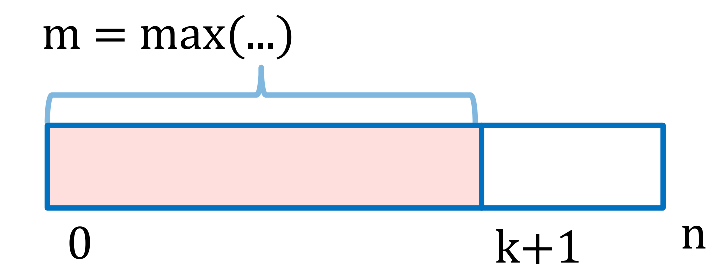
…

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

Top:



Bottom:



Tricky because max(..) involves two sets of facts
(the “for any” and the “A[i] = m”)

Max of an Array (Problem Type 2)

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

…

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

Step 1: What facts are <u>new</u> in the bottom assertion?

Just A[k] $\leq$ m

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

Max of an Array (Problem Type 2)

```
while (k < n) {
  {{ (A[j] ≤ m for any 0 ≤ j < k) and A[i] = m and 0 ≤ i < k < n }}
  if (A[k] <= m) {
    // we're good!
  } else {
    // uh oh! what now ??
  }
  {{ ((A[j] ≤ m for any 0 ≤ j < k+1) and A[i] = m  and 0 ≤ i < k+1 ≤ n }}
  k = k + 1n;
}
```

Step 2: What do we do if $A[k] > m$ does not hold?

We must change m so that $A[k] \le m$ holds again

But we also need to $A[i] = m$ (and $0 \le i < k+1$) to still hold

How do we do that?

Max of an Array (Problem Type 2)

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

```
let k: bigint = 1n;
let i: bigint = 0n;
let m: bigint = A[0];
```

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

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

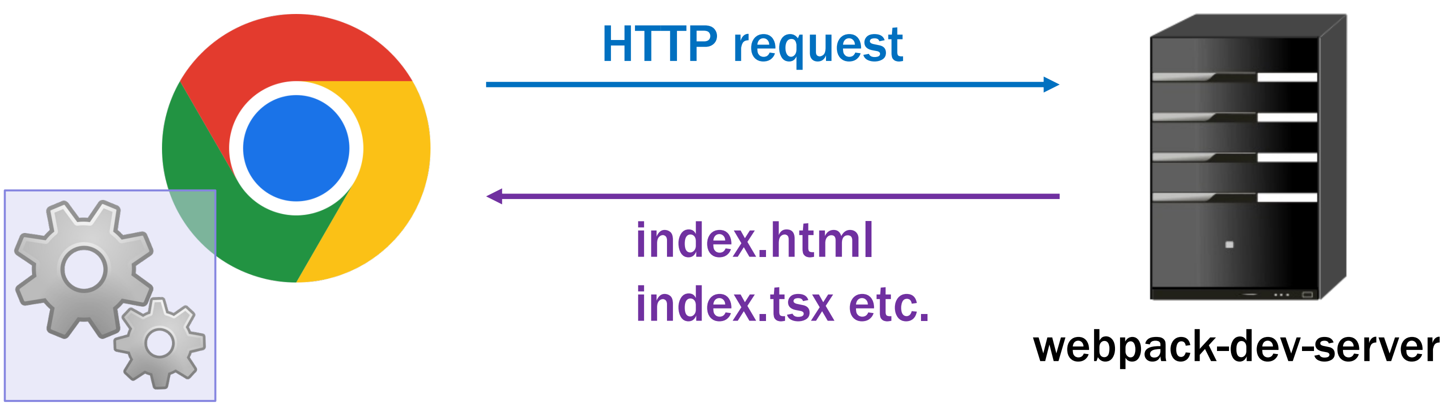
{{ **Post**: (A[j] ≤ m for any 0 ≤ j < n_ and A[i] = m and 0 ≤ i < n }}

```
return m;
```

Servers & Routes

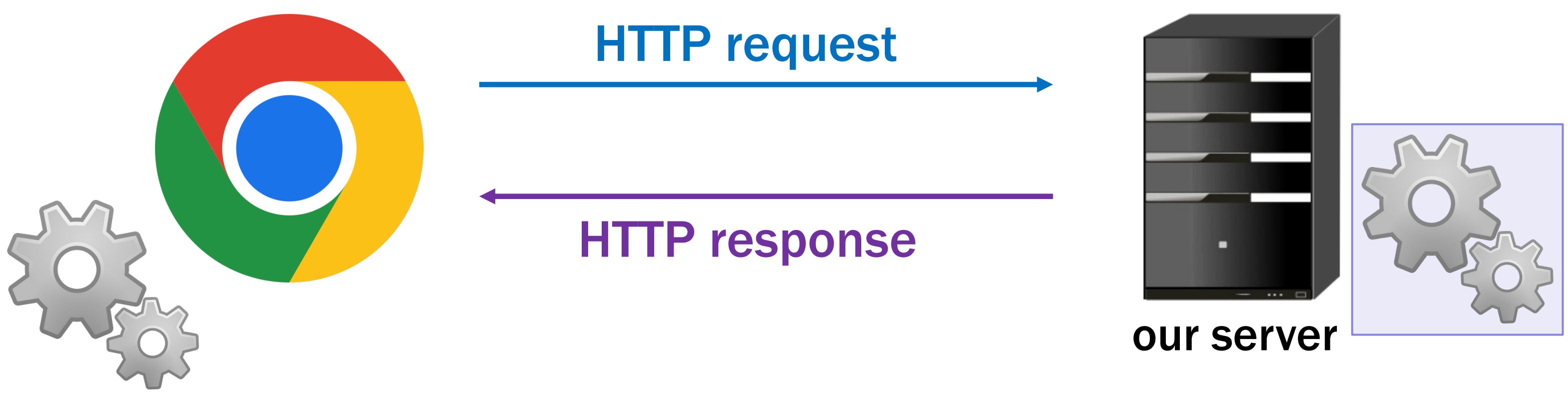
Client-Side JavaScript

- Code so far has run inside the browser
  - webpack-dev-server handles HTTP requests
  - sends back our code to the browser
- Browser executes the code of `index.tsx`
  - calls `root.render` to produce the UI



Server-Side JavaScript

- Can run code in the server as well
  - allows us to store data on the server instead
  - “node” executes the code of `index.ts`
- Start writing server-side code in HW Chatbot
  - will have code in **<u>both</u>** browser and server in HW Squares/Final



HTTP Terminology

- HTTP request includes
  - method: GET or POST (for us)

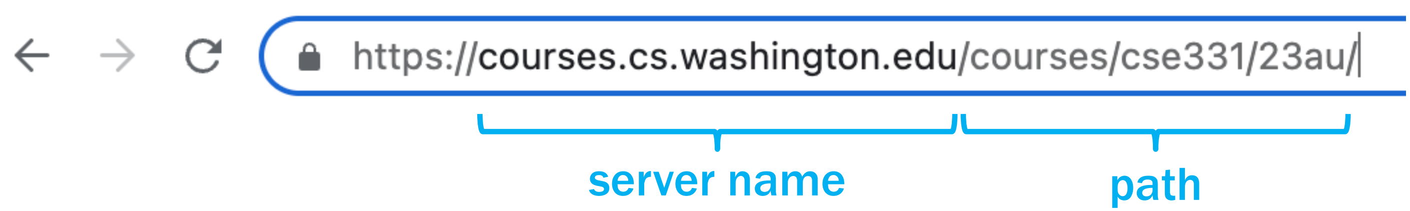
    GET is used to *read* data stored on the server (cacheable)

    POST is used to *change* data stored on the server
  - URL: path and query parameters

    can include query parameters
  - body (for POST only)

    useful for sending large or non-string data with the request

- Browser issues a GET request when you type URL



HTTP Terminology

- HTTP response includes
  - **status code**: 200 (ok), 400-99 (client error), or 500-99 (server error)

    was the server able to respond
  - **content type**: text/HTML or application/JSON (for us)

    what sort of data did the server send back
  - **content**

    in format described by the Content Type

- Browser expects HTML to display in the page
  - we will send JSON data back to our code in the browser

Custom Server

- **Create a custom server as follows:**

```
const F = (req: SafeRequest, res: SafeResponse): void => {
  …
}

const app = express();
app.get("/foo", F);
app.listen(8080);
```

  - **request for** `http://localhost:8080/foo` **will call** `F`
  - **mapping from "**`/foo`**" to** `F` **is called a "route"**
  - **can have as many routes as we want (with different URLs)**

`SafeRequest` **is an alias of** `Request<..>` **with proper type parameters filled in**

Custom Server

- **Query parameters (e.g.,** `?name=Fred`**) in** `SafeRequest`

```
const F = (req: SafeRequest, res: SafeResponse): void => {
  const name: string|undefined = req.query.name;
  if (name === undefined) {
    res.status(400).send("Missing 'name'");
    return;
  }
  …  // name was provided
}
```

type is more complicated…
parameters can be repeated

  - set status to 400 to indicate a client error (Bad Request)
  - set status to 500 to indicate a server error
  - default status is 200 (OK)

Custom Server

- **Query parameters (e.g., `?name=Fred`) in `SafeRequest`**

```
const F = (req: SafeRequest, res: SafeResponse): void => {
  const name: string|undefined = first(req.query.name);
  if (name === undefined) {
    res.status(400).send("Missing 'name'");
    return;
  }
  …  // name was provided
}
```

  - **set status to 400 to indicate a client error (Bad Request)**
  - **set status to 500 to indicate a server error**
  - **default status is 200 (OK)**

Custom Server

- **Query parameters (e.g., `?name=Fred`) in `SafeRequest`**

```
const F = (req: SafeRequest, res: SafeResponse): void => {
  const name: string|undefined = req.query.name;
  if (name === undefined) {
    res.status(400).send("Missing 'name'");
    return;
  }
  res.send({message: `Hi, ${name}`});
}
```

  - `send` **of string returned as text/HTML**
  - `send` **of record returned as application/JSON**

Example App from Section Chatbot

Animal Trivia

Question	What is your favorite color?
Answer	

Submit

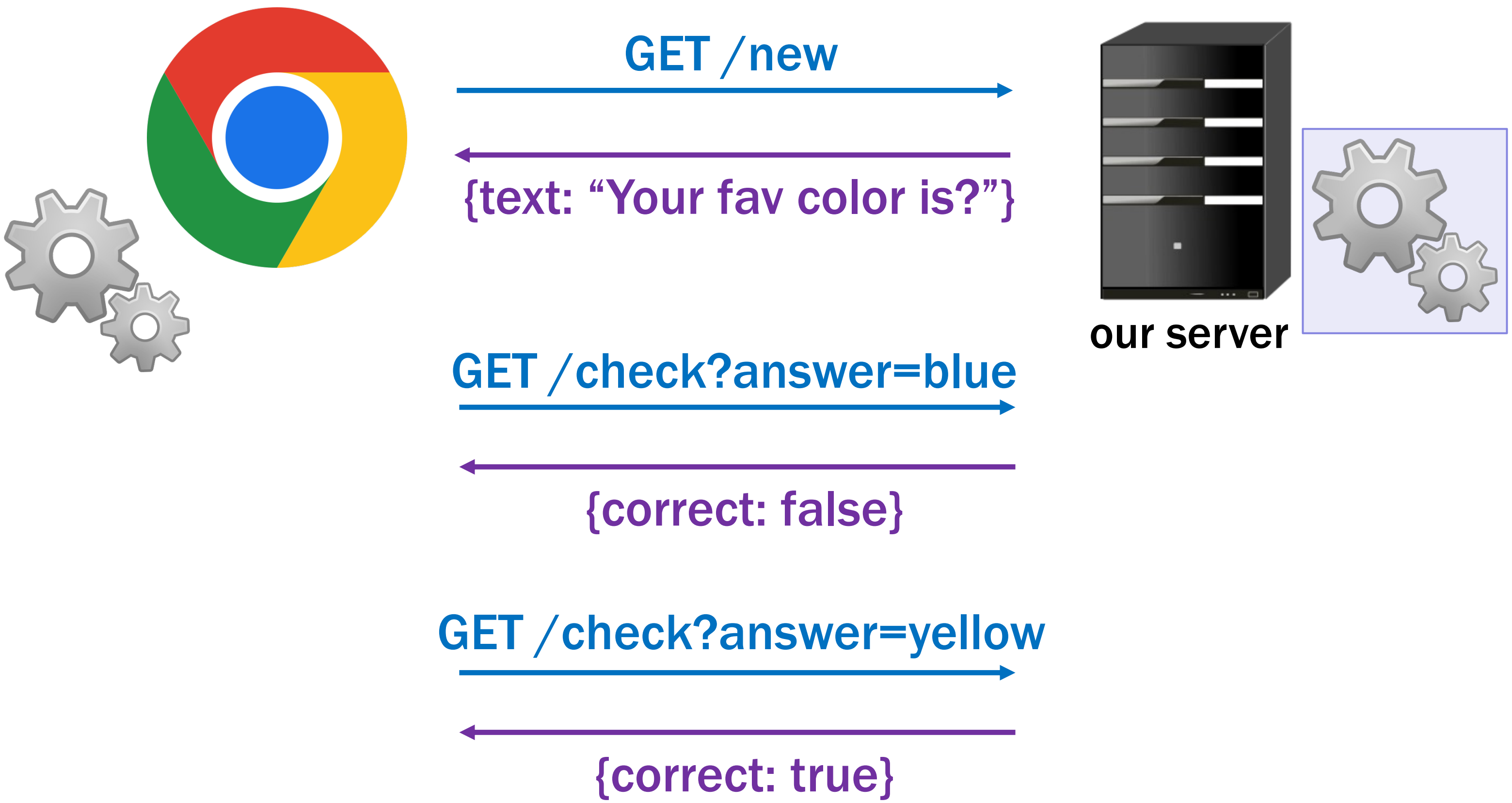
User types “blue” and presses “Submit”...

Sorry, your answer was incorrect.

New Question

Server-Side JavaScript

- Apps will make sequence of requests to server



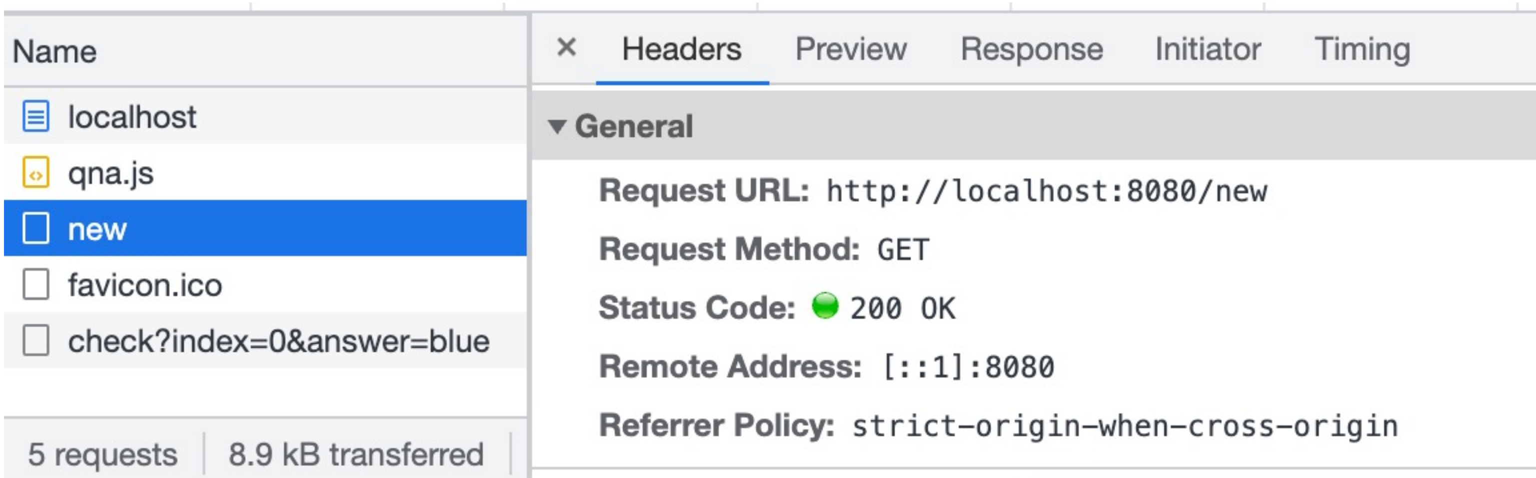
“Network” Tab Shows Requests

Name	Status
localhost	200
qna.js	200
new	200
favicon.ico	200
check?index=0&answer=blue	304

- **Shows every request to the server**
  - **first request loads the app (as usual)**
  - “`new`” **is a request to get a question**
  - “`check?index=0&answer=blue`” **is a request to check answer**

- **Click on a request to see details…**

“Network” Tab Shows Request & Response



JSON

- JavaScript Object Notation
  - text description of JavaScript object
  - allows strings, numbers, null, arrays, and records

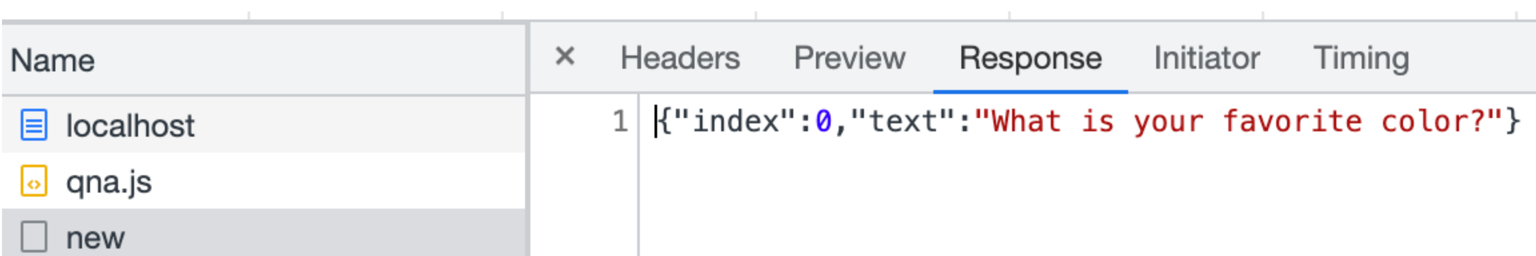
    no undefined and no instances of classes

    no ‘..’ (single quotes), only “..”

    requires quotes around keys in records
  - another tree!

- Translation into string done *automatically* by send

```
res.send({index: 0, text: ’What is your …?’});
```



Testing Server-Side TypeScript

- A route calls an ordinary function
- Testing is the same as on the client side
  - write unit tests in `X_test.ts` files
  - run then using `npm run test`
- Libraries help set up Request & Response for tests
  - can check the status returned was correct

    e.g., 200 or 400
  - can check the response body was correct

    e.g., “Missing ‘name’” or {message: “Hi, Fred”}

Testing Server-Side TypeScript

- A route calls an ordinary function

- Client- and server-side code is made up of functions
  - server functions handles requests for specific URLs
  - client functions draw data, create requests, etc.
  - test (and code review) each one

- **Key Point**: unit test each function thoroughly
  - often hard to figure which part caused the failure

    failure in the client could be due to a bug in the server
  - debugging that will be painful
  - need a higher standard of correctness in a larger app

    much easier to debug failing tests than errors in the app

Functions with Mutations

Specifying Functions that Mutate

- **Our functions so far have not mutated anything**

  makes things *much* simpler!

- **Cannot yet write a spec for sorting an array**
  - **could return a sorted version of the array**
  - **but cannot say that we change the array to be sorted**

- **Need some new tags to describe that…**

Specifying Functions that Mutate

- By default, no parameters are mutated
  - must *explicitly* say that mutation is possible (default not)

```
/**
 * Reorders A so the numbers are in increasing order
 * @param A array of integers to be sorted
 * @modifies A
 * @effects A contains the same numbers but now in
 *   increasing order
 */
const quickSort = (A: bigint[]): void => { .. };
```

  - anything that might be changed is listed in `@modifies`

    not a promise to modify it — `A` could already be sorted!

    a shorter modifies list is a **stronger** specification

Specifying Functions that Mutate

- By default, no parameters are mutated
  - must *explicitly* say that mutation is possible (default not)

```
/**
 * Reorders A so the numbers are in increasing order
 * @param A array of integers to be sorted
 * @modifies A
 * @effects A contains the same numbers but now in
 *   increasing order
 */
const quickSort = (A: bigint[]): void => { .. };
```

  - `@effects` gives promises about result after mutation

    like `@returns` but for mutated values, not return value

    this returns void, so no `@returns`

Mutating Arrays

- **Assigning to array elements changes known state**

{{ A[j – 1] < A[j] for any $1 \le j \le 5$ }}

```
A[0] = 100;
```

{{ A[j – 1] < A[j] for any $2 \le j \le 5$ and A[0] = 100 }}

- **Can add to the end of an array**

```
A.push(100);
```

{{ $A = A_0 \mathbin{+\!\!+} [100]$ }}

- **Can remove from the end of an array**

```
A.pop();
```

{{ $A = A_0[0 \,..\, n - 2]$ }}

A has one fewer element than before

Example Mutating Function

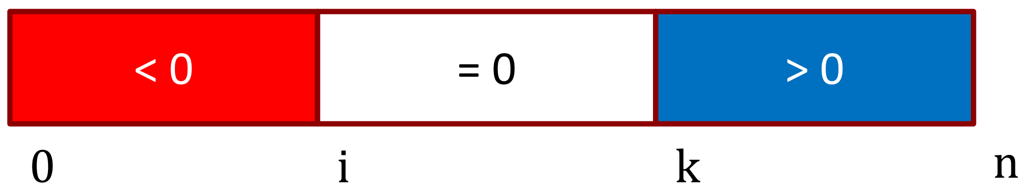
- Reorder an array so that
  - negative numbers come first, then zeros, then positives

    (not necessarily fully sorted)

```
/**
 * Reorders A into negatives, then 0s, then positive
 * @modifies A
 * @effects leaves same integers in A but with
 *   A[j] < 0 for 0 <= j < i
 *   A[j] = 0 for i <= j < k
 *   A[j] > 0 for k <= j < n
 * @returns the indexes (i, k) above
 */
const sortPosNeg = (A: bigint[]): [bigint,bigint] =>
```

Example: Sorting Negative, Zero, Positive

```
// @effects leaves same numbers in A but with
//   A[j] < 0 for 0 <= j < i
//   A[j] = 0 for i <= j < k
//   A[j] > 0 for k <= j < n
```



Let's implement this…

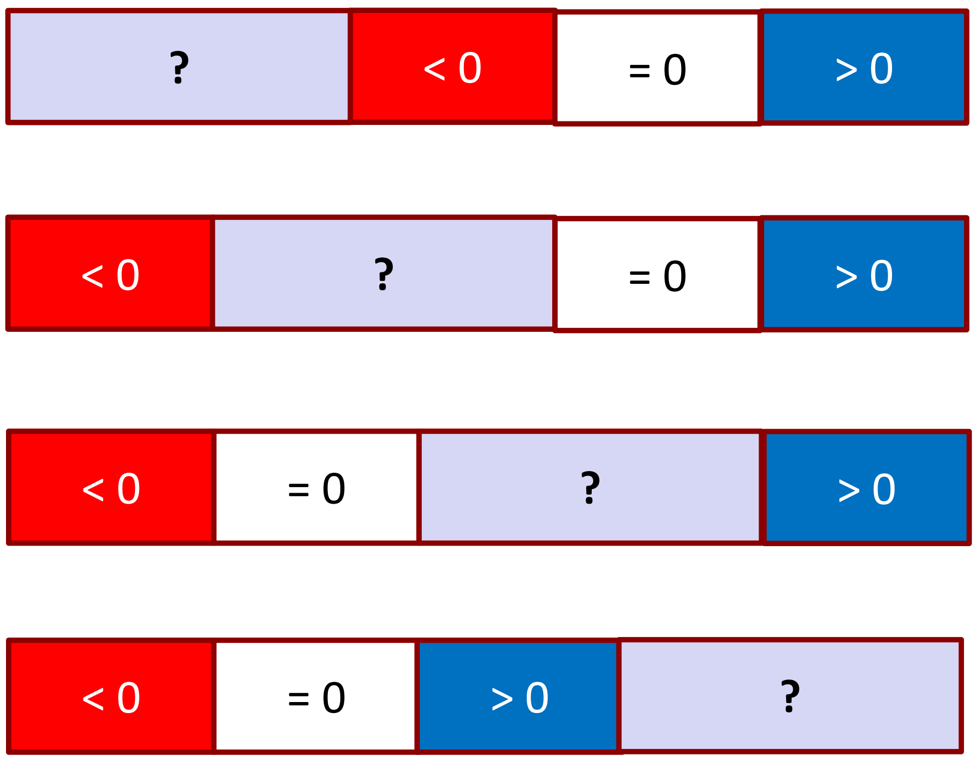
- what was our heuristic for guessing an invariant?
- weaken the postcondition

Example: Sorting Negative, Zero, Positive

How should we weaken this for the invariant?

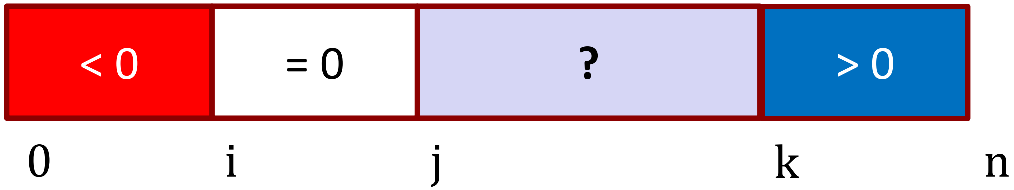
- needs allow elements with *unknown* values

    initially, we don't know anything about the array values



Example: Sorting Negative, Zero, Positive

Our Invariant:



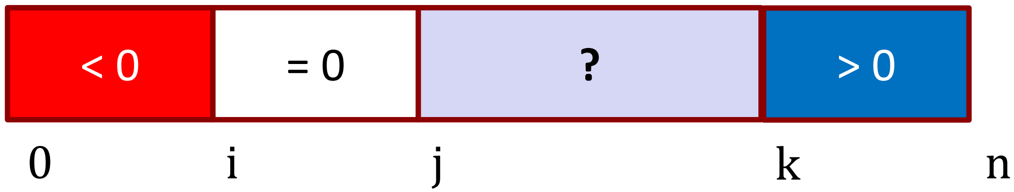
$A[\ell] < 0$ for any $0 \leq \ell < i$

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

(no constraints on $A[\ell]$ for $j \leq \ell < k$)

$A[\ell] > 0$ for any $k \leq \ell < n$

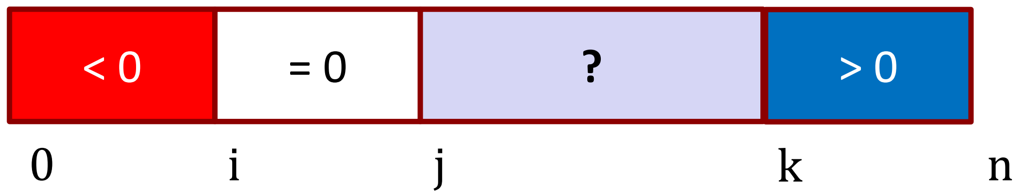
Example: Sorting Negative, Zero, Positive



- Let’s try figuring out the code (problem type 2)
  - on homework, this would be type 3 (check correctness)

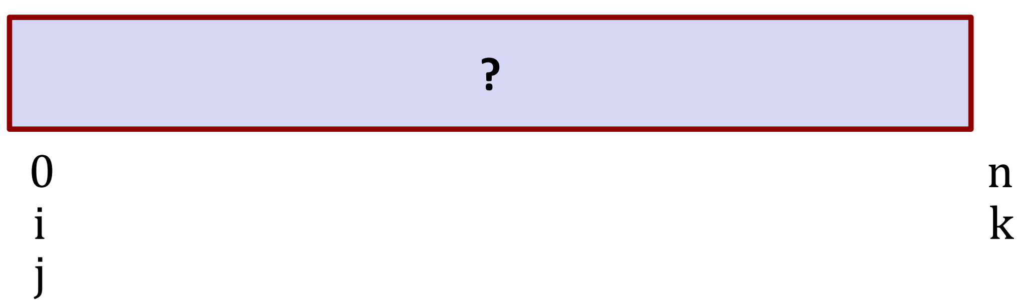
- Figure out the code for
  - how to initialize
  - when to exit
  - loop body

Example: Sorting Negative, Zero, Positive

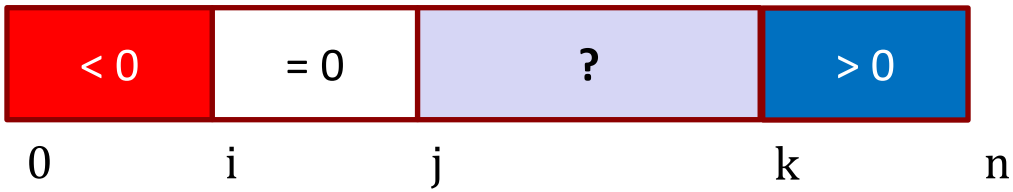


- **Will have variables** i, j, **and** k **with** $i \leq j < k$

- **How do we set these to make it true initially?**
  - **we start out not knowing anything about the array values**
  - **set** $i = j = 0$ **and** $k = n$

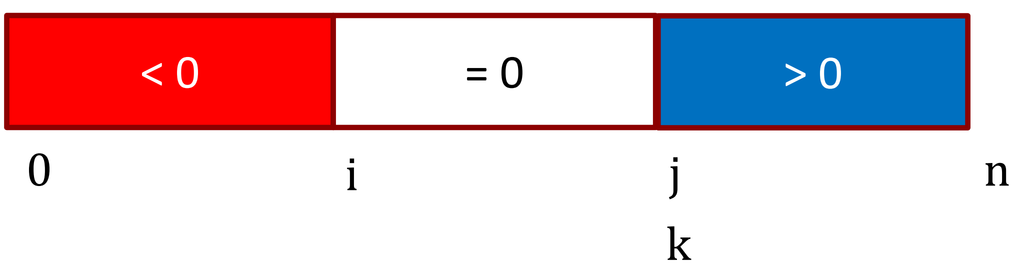


Example: Sorting Negative, Zero, Positive



- **Set** $\mathrm{i} = \mathrm{j} = 0$ **and** $\mathrm{k} = \mathrm{n}$ **to make this hold initially**

- **When do we exit?**
  - **purple is empty if** $\mathrm{j} = \mathrm{k}$



Sort Positive, Zero, Negative

```
let i: bigint = 0n;
let j: bigint = 0;
let k: bigint = A.length;
```

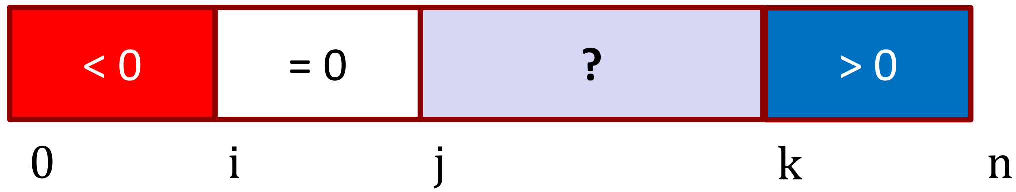
{{ **Inv**: $A[\ell] < 0$ for any $0 \le \ell < i$ and $A[\ell] = 0$ for any $i \le \ell < j$
$A[\ell] > 0$ for any $k \le \ell < n$ and $0 \le i \le j \le k \le n$}}

```
while (j < k) {
  ...
}
```

{{ $A[\ell] < 0$ for any $0 \le \ell < i$ and $A[\ell] = 0$ for any $i \le \ell < j$
$A[\ell] > 0$ for any $j \le \ell < n$ }}

```
return [i, j];
```

Example: Sorting Negative, Zero, Positive

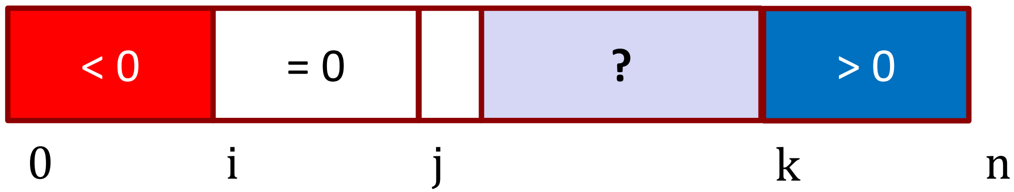


- How do we make progress?
  - try to increase j by 1 or decrease k by 1

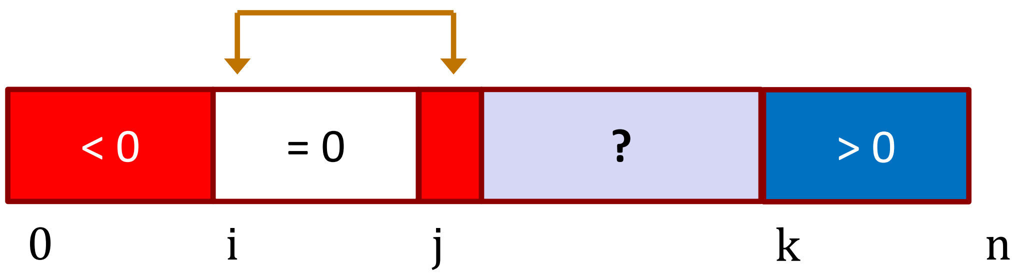
- Look at $A[j]$ and figure out where it goes

- What to do depends on $A[j]$
  - could be < 0, $= 0$, or > 0

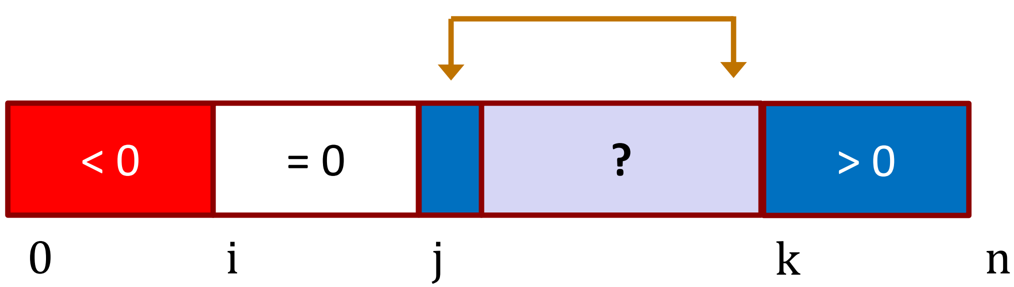
Example: Sorting Negative, Zero, Positive



Set $j = j_0 + 1$



Swap $A[i]$ **and** $A[j]$
Set $i = i_0 + 1$
and $j = j_0 + 1$



Swap $A[j]$ **and** $A[k-1]$
Set $k = k_0 - 1$

Sort Positive, Zero, Negative

{{ **Inv**: $A[\ell] < 0$ for any $0 \le \ell < i$ and $A[\ell] = 0$ for any $i \le \ell < j$
$A[\ell] > 0$ for any $k \le \ell < n$ and $0 \le i \le j \le k \le n$ }}

```
while (j !== k) {
  if (A[j] === 0n) {
    j = j + 1n;
  } else if (A[j] < 0n) {
    swap(A, i, j);
    i = i + 1n;
    j = j + 1n;
  } else {
    swap(A, j, k);
    k = k - 1n;
  }
}
```

Combine forward and backward reasoning to double check correctness.

Sort Positive, Zero, Negative

{{ **Inv**: $A[\ell] < 0$ for any $0 \le \ell < i$ and $A[\ell] = 0$ for any $i \le \ell < j$
$A[\ell] > 0$ for any $k \le \ell < n$ }}

```
while (j !== k) {
  …
  } else if (A[j] < 0n) {
```

{{ $A[\ell] < 0$ for any $0 \le \ell < i$ and $A[\ell] = 0$ for any $i \le \ell < j$
$A[\ell] > 0$ for any $k \le \ell < n$ and $0 \le i \le j \le k \le n$ and $\mathbf{A[j] < 0}$ }}

```
    swap(A, i, j);
    i = i + 1n;
    j = j + 1n;
```

{{ $A[\ell] < 0$ for any $0 \le \ell < i$ and $A[\ell] = 0$ for any $i \le \ell < j$
$A[\ell] > 0$ for any $k \le \ell < n$ and $0 \le i \le j \le k \le n$ }}

```
  }
  …
```

Sort Positive, Zero, Negative

{{ **Inv**: $A[\ell] < 0$ for any $0 \le \ell < i$ and $A[\ell] = 0$ for any $i \le \ell < j$
$A[\ell] > 0$ for any $k \le \ell < n$ }}

```
while (j !== k) {
  …
  } else if (A[j] < 0n) {
```

{{ $A[\ell] < 0$ for any $0 \le \ell < i$ and $A[\ell] = 0$ for any $i \le \ell < j$
$A[\ell] > 0$ for any $k \le \ell < n$ and $A[j] < 0$ }}

```
    swap(A, i, j);
```

{{ $A[\ell] < 0$ for any $0 \le \ell < i+1$ and $A[\ell] = 0$ for any $i+1 \le \ell < j+1$
$A[\ell] > 0$ for any $k \le \ell < n$ and $0 \le i+1 \le j+1 \le k \le n$ }}

```
    i = i + 1n;
    j = j + 1n;
```

{{ $A[\ell] < 0$ for any $0 \le \ell < i$ and $A[\ell] = 0$ for any $i \le \ell < j$
$A[\ell] > 0$ for any $k \le \ell < n$ and $0 \le i \le j \le k \le n$ }}

```
  }
  …
```

Sort Positive, Zero, Negative

{{ $A[\ell] < 0$ for any $0 \le \ell < i$ and $A[\ell] = 0$ for any $i \le \ell < j$
$A[\ell] > 0$ for any $k \le \ell < n$ and $0 \le i \le j \le k \le n$ and $A[j] < 0$ }}

```
swap(A, i, j);
```

{{ $A[\ell] < 0$ for any $0 \le \ell < i{+}1$ and $A[\ell] = 0$ for any $i{+}1 \le \ell < j{+}1$
$A[\ell] > 0$ for any $k \le \ell < n$ and $0 \le i{+}1 \le j{+}1 \le k \le n$ }}

Easiest to stop here since this is a function call. (Need to use its spec.)

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

New facts are $A[i] < 0$ and $A[j] = 0$

Initially have $A[i] = 0$ and $A[j] < 0$

Swapping them gives what we want.

Other 2 cases are similar... (Exercise)