CSE 331

Arrays

James Wilcox and Kevin Zatloukal

Indexing

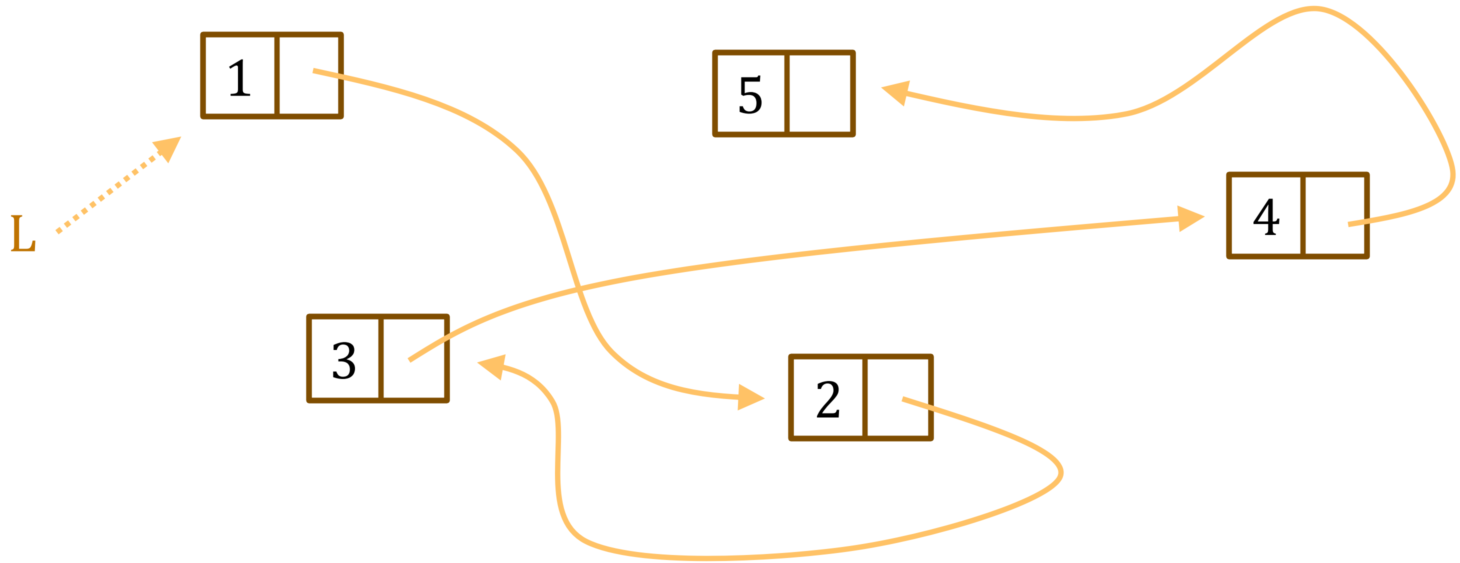
$$\mathrm{at} : (\mathrm{List}, \mathbb{N}) \to \mathbb{Z}$$

$$\begin{array}{ll}
\mathrm{at}(\mathrm{nil}\ , n) & := \ \mathrm{undefined} \\
\mathrm{at}(x :: L\ , 0) & := \ x \\
\mathrm{at}(x :: L\ , n+1) & := \ \mathrm{at}(L, n)
\end{array}$$

- **Retrieve an element of the list by <u>index</u>**
  - **use "$L[j]$" as an abbreviation for $\mathrm{at}(j, L)$**

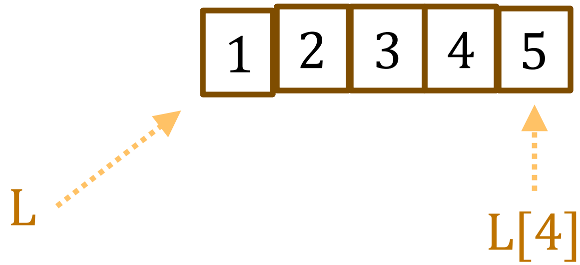
- **Not an efficient operation on lists…**

Linked Lists in Memory



- **Must follow the "next" pointers to find elements**
  - $\mathrm{at}(L, n)$ **is an** $O(n)$ **operation**
  - **no faster way to do this**

Faster Implementation of at

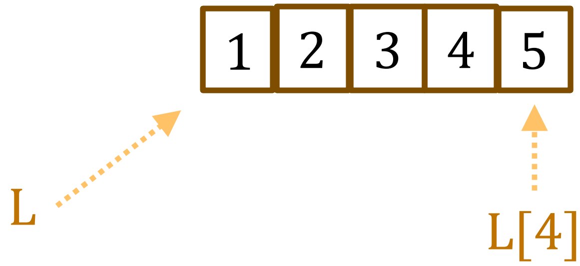


- **Alternative: store the elements next to each other**
  - **can find the n-th entry by arithmetic:**

location of L[4] = (location of L) + 4 * sizeof(data)

- **Resulting data structure is an array**

Faster Implementation of at



- Resulting data structure is an **array**

- Efficient to read $\mathrm{L}[\mathrm{i}]$

- Inefficient to…
  - insert elements anywhere but the end
  - write operations with an immutable ADT
  - trees can do <u>all of this</u> in $O(\log n)$ time

Access By Index

- Easily access both $\mathrm{L}[0]$ and $\mathrm{L}[\mathrm{n}\text{-}1]$, where $\mathrm{n} = \mathrm{len}(\mathrm{L})$
  - can process a list in either direction

- “With great power, comes great responsibility”

  — the Peter Parker Principle

- Whenever we write “$\mathrm{A}[\mathrm{j}]$”, we must check $0 \leq \mathrm{j} < \mathrm{n}$
  - 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

Recall: Sum List With a Loop

$$\text{sum-acc}(\text{nil}, r) := r$$
$$\text{sum-acc}(x :: L, r) := \text{sum-acc}(L, x + r)$$

- Tail recursive version is a loop

```
const sum = (S: List<bigint>): bigint => {
  let r = 0;
  // Inv: sum(S_0) = r + sum(S)
  while (S.kind !== "nil") {
    r = S.hd + r;
    S = S.tl;
  }
  return r;
};
```

Change to a version that uses indexes...

Sum List by Index

- Change to using an array and accessing by index

```
const sum = (S: Array<bigint>): bigint => {
  let r = 0;
  let j = 0;
  // Inv: …
  while (j !== S.length) { // … S.kind !== "nil"
    r = S[j] + r;          // … r = S.hd + r
    j = j + 1;             // … S = S.tl
  }
  return r;
};
```

Note that **S** is no longer changing

Sum List by Index

$$\text{sum-acc} : (\text{List}, \mathbb{N}, \mathbb{Z}) \to \mathbb{Z}$$

$$\begin{array}{lll} \text{sum-acc}(S, j, r) & := r & \textbf{if } j = \text{len}(S) \\ \text{sum-acc}(S, j, r) & := \text{sum-acc}(S, j+1, S[j] + r) & \textbf{if } j \neq \text{len}(S) \end{array}$$

- **Change to using an array and accessing by index**

```
const sum = (S: Array<bigint>): bigint => {
  let r = 0;
  let j = 0;
  // Inv: …
  while (j !== S.length) {
    r = S[j] + r;
    j = j + 1;
  }
  return r;
};
```

Sublists

- **Use indexes to refer to a section of a list (a "sublist"):**

$$\text{sublist} : (\text{List}, \mathbb{Z}, \mathbb{Z}) \to \mathbb{Z}$$

$$\begin{aligned} \text{sublist}(L, i, j) &:= \text{nil} && \textbf{if } j < i \\ \text{sublist}(L, i, j) &:= L[i] :: \text{sublist}(L, i+1, j) && \textbf{if } i \leq j \end{aligned}$$

- **Useful for *reasoning* about lists and indexes**

- **This includes <u>both</u> $L[i]$ and $L[j]$**

$$\begin{aligned} \text{sublist}(L, 0, 2) &= L[0] :: \text{sublist}(L, 1, 2) && \textbf{def of } \text{sublist } (\textbf{since } 0 \leq 2) \\ &= L[0] :: L[1] :: \text{sublist}(L, 2, 2) && \textbf{def of } \text{sublist } (\textbf{since } 1 \leq 2) \\ &= L[0] :: L[1] :: L[2] :: \text{sublist}(L, 3, 2) && \textbf{def of } \text{sublist } (\textbf{since } 2 \leq 2) \\ &= L[0] :: L[1] :: L[2] :: \text{nil} && \textbf{def of } \text{sublist } (\textbf{since } 3 < 2) \\ &= [L[0], L[1], L[2]] \end{aligned}$$

Sublists

- Use indexes to refer to a section of a list (a "sublist"):

$$\text{sublist} : (\text{List}, \mathbb{Z}, \mathbb{Z}) \to \mathbb{Z}$$

$$\begin{array}{lll} \text{sublist}(L, i, j) & := \text{nil} & \textbf{if } j < i \\ \text{sublist}(L, i, j) & := L[i] :: \text{sublist}(L, i+1, j) & \textbf{if } i \leq j \end{array}$$

- The sublist is empty when the **range** is empty

$$\text{sublist}(L, 3, 2) = \text{nil}$$

  - weird-looking example that comes up a lot:

$$\text{sublist}(L, 0, \text{-}1) = \text{nil}$$

  - not an array out of bonds error! (this is math, not Java)

Sublists

$$\text{sublist} : (\text{List}, \mathbb{Z}, \mathbb{Z}) \to \mathbb{Z}$$

$$\begin{aligned} \text{sublist}(L, i, j) &:= \text{nil} && \textbf{if } j < i \\ \text{sublist}(L, i, j) &:= L[i] :: \text{sublist}(L, i+1, j) && \textbf{if } i \le j \end{aligned}$$

- **Will use "$L[i .. j]$" as shorthand for "$\text{sublist}(L, i, j)$"**
  - **again, using an operator for most common operations**

- **Some useful facts about sublists:**

$$L = L[0 .. \text{len}(L)\text{-}1]$$

$$L[i .. j] = L[i .. k] +\!\!+ L[k+1 .. j]$$ **for any** k **with** $i - 1 \le k \le j$ (and $0 \le i \le j < n$)

Sum List by Index

$$\begin{array}{lll} \text{sum-acc}(S, j, r) & := r & \textbf{if } j = \text{len}(S) \\ \text{sum-acc}(S, j, r) & := \text{sum-acc}(S, j+1, S[j] + r) & \textbf{if } j \neq \text{len}(S) \end{array}$$

- **Change to using an array and accessing by index**

```
const sum = (S: Array<bigint>): bigint => {
  let r = 0;
  let j = 0;
  // Inv: … ?? …
  while (j != S.length) {
    r = S[j] + r;
    j = j + 1;
  }
  return r;
};
```

Still need to fill in Inv…

Need a version using indexes.

Recall: Sum List With a Loop

$$\begin{aligned}
&\text{sum-acc}(S, j, r) &&:= r && \textbf{if } j = \text{len}(S) \\
&\text{sum-acc}(S, j, r) &&:= \text{sum-acc}(S, j{+}1, S[j] + r) && \textbf{if } j \neq \text{len}(S)
\end{aligned}$$

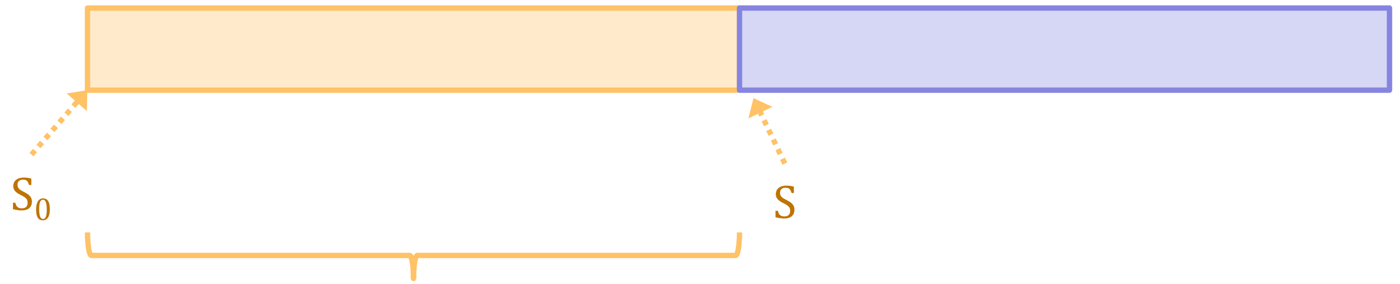
- **Tail recursive version is a loop**

```
const sum = (S: List<bigint>): bigint => {
  let r = 0;
  // Inv: sum(S₀) = r + sum(S)
  while (S.kind !== "nil") {
    r = S.hd + r;
    S = S.tl;
  }
  return r;
};
```

Inv says $\text{sum}(S_0)$ is r plus sum of rest (S)

Not the most explicit way of explaining "r"...

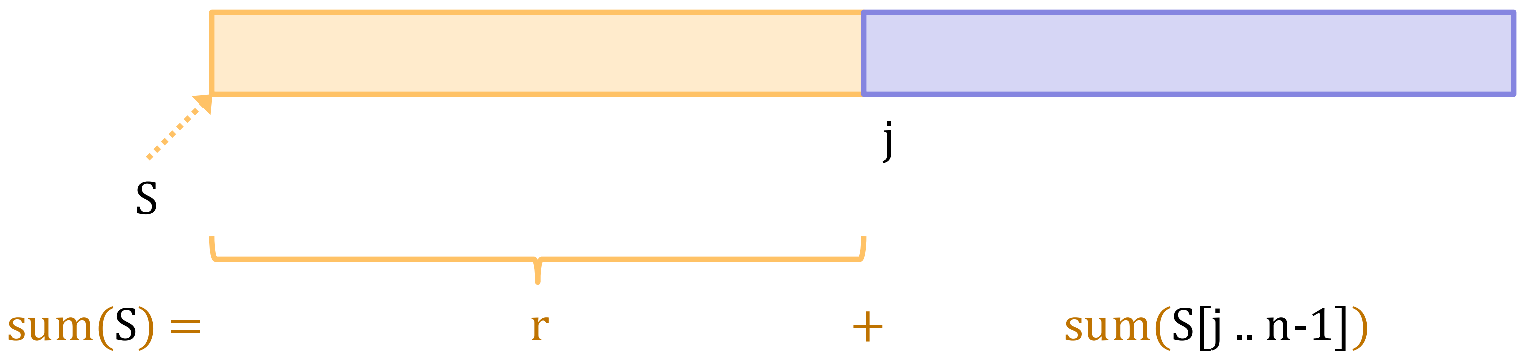
Recall: Sum List With a Loop



$$\mathrm{sum}(S_0) = r + \mathrm{sum}(S)$$

- "r" contains sum of the part of the list *seen so far*
- Can explain this more simply with indexes…
  - no longer need to move S

Using Sublists With Loops



- **Sum is the part in "r" plus the part left in S[j .. n-1]**

- **What sum is in "r"?**

$$r = \text{sum}(S[0 \,..\, j-1])$$

  – **we can use just this as our invariant! (it's all we need)**

Using Sublists With Loops

- Array version uses access by index

```
const sum = (S: Array<bigint>): bigint => {
  let r = 0;
  let j = 0;
  // Inv: r = sum(S[0 .. j-1])
  while (j != S.length) {
    r = S[j] + r;
    j = j + 1;
  }
  return r;
};
```

Are we sure this is right?

Let's think it through...

Sum of an Array

```
const sum = (S: Array<bigint>): bigint => {
  let r = 0;
  let j = 0;
  {{ r = 0 and j = 0 }}
  {{ Inv: r = sum(S[0 .. j-1]) }}
  while (j != S.length) {
    r = S[j] + r;
    j = j + 1;
  }
  return r;
};
```

Does Inv hold initially?

sum(S[0 .. j-1])
= sum(S[0 .. -1]) since $j = 0$
= sum([])
= 0 def of sum
= r

Sum of an Array

```
const sum = (S: Array<bigint>): bigint => {
  let r = 0;
  let j = 0;
  {{ Inv: r = sum(S[0 .. j-1]) }}
  while (j != S.length) {
    r = S[j] + r;
    j = j + 1;
  }
  {{ r = sum(S[0 .. j-1]) and j = len(S) }}
  {{ r = sum(S) }}
  return r;
};
```

Does the postcondition hold?

$$
\begin{aligned}
r &= \mathrm{sum}(S[0 \,..\, j\text{-}1]) \\
&= \mathrm{sum}(S[0 \,..\, \mathrm{len}(S)\text{-}1]) && \textbf{since } j = \mathrm{len}(S) \\
&= \mathrm{sum}(S)
\end{aligned}
$$

Sum of an Array

```
const sum = (S: Array<bigint>): bigint => {
  let r = 0;
  let j = 0;
  {{ Inv: r = sum(S[0 .. j-1]) }}
  while (j != S.length) {
    {{ r = sum(S[0 .. j-1]) and j ≠ len(S) }}
    r = S[j] + r;
    j = j + 1;
    {{ r = sum(S[0 .. j-1]) }}
  }
  return r;
};
```

Sum of an Array

```
const sum = (S: Array<bigint>): bigint => {
  let r = 0;
  let j = 0;
  {{ Inv: r = sum(S[0 .. j-1]) }}
  while (j != S.length) {
    {{ r = sum(S[0 .. j-1]) and j ≠ len(S) }}
    r = S[j] + r;
    {{ r = sum(S[0 .. j]) }}
    j = j + 1;
    {{ r = sum(S[0 .. j-1]) }}
  }
  return r;
};
```

Sum of an Array

```
const sum = (S: Array<bigint>): bigint => {
  let r = 0;
  let j = 0;
  {{ Inv: r = sum(S[0 .. j-1]) }}
  while (j != S.length) {
    {{ r = sum(S[0 .. j-1]) and j ≠ len(S) }}
    {{ S[j] + r = sum(S[0 .. j]) }}
    r = S[j] + r;
    {{ r = sum(S[0 .. j]) }}
    j = j + 1;
    {{ r = sum(S[0 .. j-1]) }}
  }
  return r;
};
```

Sum of an Array

```
const sum = (S: Array<bigint>): bigint => {
  let r = 0;
  let j = 0;
  {{ Inv: r = sum(S[0 .. j-1]) }}
  while (j != S.length) {
    {{ r = sum(S[0 .. j-1]) and j ≠ len(S) }}
    {{ S[j] + r = sum(S[0 .. j]) }}
    r = S[j] + r;
    {{ r = sum(S[0 .. j]) }}
    j = j + 1;
    {{ r = sum(S[0 .. j-1]) }}
  }
  return r;
};
```

Is this valid?

Sum of an Array

{{ $r = \text{sum}(S[0 .. j-1])$ and $j \neq \text{len}(S)$ }}
{{ $S[j] + r = \text{sum}(S[0 .. j])$ }}

$S[j] + r$
$= S[j] + \text{sum}(S[0 .. j-1])$ **since** $r = \text{sum}(S[0 .. j-1])$
$= \text{sum}(S[0 .. j-1]) + S[j]$
$= \text{sum}(S[0 .. j-1]) + \text{sum}([S[j]])$ **def of** sum
$= \text{sum}(S[0 .. j-1]) + \text{sum}(S[j .. j])$

Sum of an Array

{{ r = sum(S[0 .. j-1]) and j ≠ len(S) }}
{{ S[j] + r = sum(S[0 .. j]) }}

S[j] + r
= S[j] + sum(S[0 .. j-1]) **since** r = sum(S[0 .. j-1])
= sum(S[0 .. j-1]) + S[j]
= sum(S[0 .. j-1]) + sum([S[j]]) **def of** sum
= sum(S[0 .. j-1]) + sum(S[j .. j])
= ...
= sum(S[0 .. j])

Sum of an Array

$\{\{\ r = \mathrm{sum}(S[0\ ..\ j\text{-}1])\ \text{and}\ j \neq \mathrm{len}(S)\ \}\}$

$\{\{\ S[j] + r = \mathrm{sum}(S[0\ ..\ j])\ \}\}$

$$
\begin{aligned}
& S[j] + r \\
& \quad = S[j] + \mathrm{sum}(S[0\ ..\ j\text{-}1]) && \textbf{since } r = \mathrm{sum}(S[0\ ..\ j\text{-}1]) \\
& \quad = \mathrm{sum}(S[0\ ..\ j\text{-}1]) + S[j] \\
& \quad = \mathrm{sum}(S[0\ ..\ j\text{-}1]) + \mathrm{sum}([S[j]]) && \textbf{def of } \mathrm{sum} \\
& \quad = \mathrm{sum}(S[0\ ..\ j\text{-}1]) + \mathrm{sum}(S[j\ ..\ j]) \\
& \quad = \ldots \\
& \quad = \mathrm{sum}(S[0\ ..\ j\text{-}1] \mathbin{+\!\!+} S[j\ ..\ j]) \\
& \quad = \mathrm{sum}(S[0\ ..\ j])
\end{aligned}
$$

- **We saw that** $\mathrm{len}(L \mathbin{+\!\!+} R) = \mathrm{len}(L) + \mathrm{len}(R)$
- **Does** $\mathrm{sum}(L \mathbin{+\!\!+} R) = \mathrm{sum}(L) + \mathrm{sum}(R)$?
  - Yes! Very similar proof by structural induction. (Call this Lemma 3)

Sum of an Array

{{ r = sum(S[0 .. j-1]) and j ≠ len(S) }}
{{ S[j] + r = sum(S[0 .. j]) }}

S[j] + r
= S[j] + sum(S[0 .. j-1]) **since** r = sum(S[0 .. j-1])
= sum(S[0 .. j-1]) + S[j]
= sum(S[0 .. j-1]) + sum([S[j]]) **def of** sum
= sum(S[0 .. j-1]) + sum(S[j .. j])
= sum(S[0 .. j-1] ++ S[j .. j]) **by Lemma 3**
= sum(S[0 .. j])

(The need to reason by induction comes up all the time.)

Sum of an Array

{{ $r - S[j-1] = \text{sum}(S[0 .. j-2])$ and $j-1 \neq \text{len}(S)$ }}
{{ $r = \text{sum}(S[0 .. j-1])$ }}

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

- **We saw that** $\text{len}(L \mathbin{+\!\!+} R) = \text{len}(L) + \text{len}(R)$
- **Does** $\text{sum}(L \mathbin{+\!\!+} R) = \text{sum}(L) + \text{sum}(R)$?
  - Yes! Very similar proof by structural induction. (Call this Lemma 3)

Sum of an Array

{{ $r - S[j\text{-}1] = \text{sum}(S[0 .. j\text{-}2])$ and $j\text{-}1 \neq \text{len}(S)$ }}
{{ $r = \text{sum}(S[0 .. j\text{-}1])$ }}

$r = S[j\text{-}1] + \text{sum}(S[0 .. j\text{-}2])$	**since** $r - S[j\text{-}1] = \text{sum}(S[0 .. j\text{-}2])$
$= \text{sum}(S[0 .. j\text{-}2]) + S[j\text{-}1]$	
$= \text{sum}(S[0 .. j\text{-}2]) + \text{sum}([S[j\text{-}1]])$	**def of** sum
$= \text{sum}(S[0 .. j\text{-}2]) + \text{sum}(S[j\text{-}1 .. j\text{-}1])$	
$= \text{sum}(S[0 .. j\text{-}2] \mathbin{+\!\!+} S[j\text{-}1 .. j\text{-}1])$	**by Lemma 3**
$= \text{sum}(S[0 .. j\text{-}1])$	

(The need to reason by induction comes up all the time.)

Linear Search of a List

$$\begin{aligned}
\text{contains}(\text{nil}, y) &:= \text{false} & & \\
\text{contains}(x :: L, y) &:= \text{true} & & \textbf{if } x = y \\
\text{contains}(x :: L, y) &:= \text{contains}(L, y) & & \textbf{if } x \neq y
\end{aligned}$$

- **Tail-recursive definition**

```
const contains =
    (S: List<bigint>, y: bigint): bigint => {
  // Inv: contains(S_0, y) = contains(S, y)
  while (S.kind !== "nil" && S.hd !== y) {
    S = S.tl;
  }
  return S.kind !== "nil";  // implies S.hd === y
};
```

Change to a version that uses indexes…

Linear Search of an Array

$$\begin{array}{lll} \text{contains}(\text{nil}, y) & := \text{false} & \\ \text{contains}(x :: L, y) & := \text{true} & \textbf{if } x = y \\ \text{contains}(x :: L, y) & := \text{contains}(L, y) & \textbf{if } x \neq y \end{array}$$

- **Change to using an array and accessing by index**

```
const contains =
    (S: Array<bigint>, y: bigint): bigint => {
  let j = 0;
  // Inv: ...
  while (j !== S.length && S[j] !== y) {
    j = j + 1;
  }
  return j !== S.length;
};
```

`S.hd` with `S` changing becomes
`S[j]` with `j` changing

What is the invariant now?

Linear Search of an Array

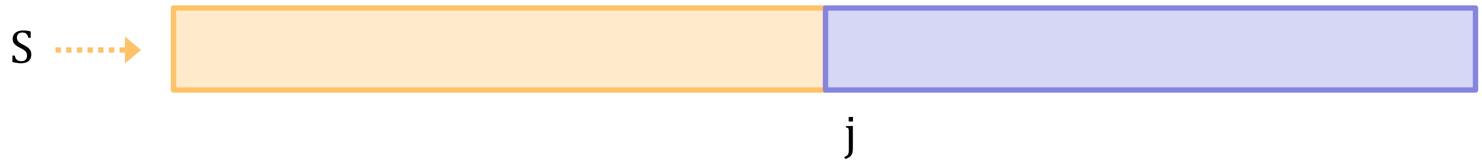
$$
\begin{array}{lll}
\text{contains}(\text{nil}, y) & := \text{false} & \\
\text{contains}(x :: L, y) & := \text{true} & \textbf{if } x = y \\
\text{contains}(x :: L, y) & := \text{contains}(L, y) & \textbf{if } x \neq y
\end{array}
$$

- **Change to using an array and accessing by index**

```
const contains =
    (S: Array<bigint>, y: bigint): bigint => {
  let j = 0;
  // Inv: contains(S, y) = contains(S[j .. n-1], y)
  while (j !== S.length && S[j] !== y) {
    j = j + 1;
  }
  return j !== S.length;
};
```

Can we explain this better?

Linear Search of an Array



contains(S, y) = contains(S[j .. n-1], y)

- What do we know about the left segment?
  - it does not contain "y"
  - that's why we kept searching

Linear Search of an Array

- **Update the invariant to be more informative**

```
const contains =
    (S: Array<bigint>, y: bigint): bigint => {
  let j = 0;
  // Inv: S[i] ≠ y for any i = 0 .. j-1
  while (j !== S.length && S[j] !== y) {
    j = j + 1;
  }
  return j !== S.length;
};
```

Facts About Sublists

- "With great power, comes great responsibility"

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

    applies to preconditions, postconditions, and intermediate assertions

- We can write facts about several elements at once:
  - this says that elements at indexes 0 .. j-1 are not y

    $\mathrm{S[i]} \neq \mathrm{y}$ for any $0 \leq \mathrm{i} < \mathrm{j}$

  - shorthand for j facts: $S[0] \neq y$, …, $S[j\text{-}1] \neq y$

Reasoning Toolkit

Description	Testing	Tools	Reasoning
no mutation	full coverage	type checker	calculation induction
local variable mutation	“	“	Floyd logic
heap state	“	“	rep invariants
arrays	“	“	for-any facts

Facts About Sublists

- "With great power, comes great responsibility"
  - since we can easily access any $\mathrm{L}[\mathrm{j}]$, may need facts about it

- We can write facts about several elements at once:
  - this says that elements at indexes 0 .. j-1 are not y

$$\mathrm{S}[\mathrm{i}] \neq \mathrm{y} \qquad \text{for any } 0 \leq \mathrm{i} < \mathrm{j}$$

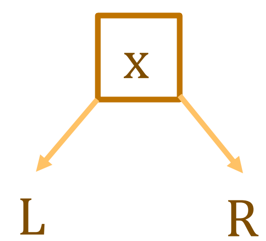
- These facts get hard to write down!
  - we will need to find ways to make this <u>easier</u>
  - a common trick is to draw pictures instead…

Visual Presentation of Facts

- Just saw this example

- But we have seen "for any" facts with BSTs...

$$\text{contains-key}(y, L) \rightarrow (y < x)$$
$$\text{contains-key}(z, R) \rightarrow (x < z)$$



  - "for any" facts are common in more complex code
  - drawing pictures is a typical coping mechanism

Recall: Linear Search of an Array

- **Let's check the correctness of this loop (w/ pictures)**

```
const contains =
    (S: Array<bigint>, y: bigint): boolean => {
  let j = 0;
  // Inv: S[k] /= y for any k = 0 .. j-1
  while (j !== S.length && S[j] !== y) {
    j = j + 1;
  }
  return j !== S.length;
};
```

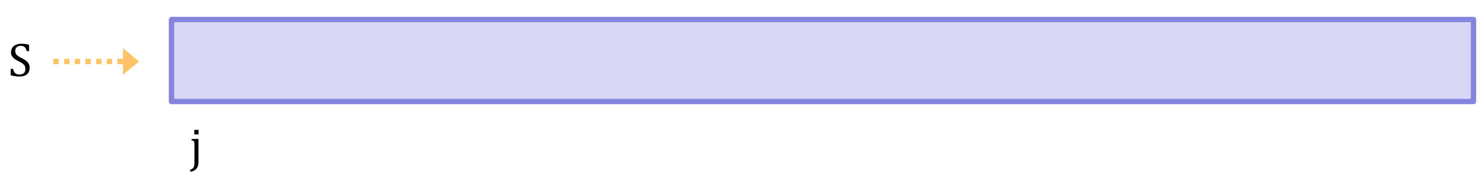
Inv: gold part contains no y

Linear Search of an Array


```
const contains =
    (S: Array<bigint>, y: bigint): boolean => {
  let j = 0;
  {{ j = 0 }}
  {{ Inv: S[i] ≠ y  for any 0 ≤ i ≤ j – 1 }}
  while (j !== S.length && S[j] !== y) {
    j = j + 1;
  }
  return j !== S.length;
};
```

What is the picture when j = 0?

Inv holds because there is no gold part.



Linear Search of an Array


```
const contains =
    (S: Array<bigint>, y: bigint): boolean => {
  let j = 0;
  {{ Inv: S[i] ≠ y  for any 0 ≤ i ≤ j – 1 }}
  while (j !== S.length && S[j] !== y) {
    {{ (S[i] ≠ y  for any 0 ≤ i ≤ j – 1) and j ≠ len(S) and S[j] ≠ y }}
    j = j + 1;
    {{ S[i] ≠ y  for any 0 ≤ i ≤ j – 1 }}
  }
  return j !== S.length;
};
```

Linear Search of an Array


```
const contains =
    (S: Array<bigint>, y: bigint): boolean => {
  let j = 0;
  {{ Inv: S[i] ≠ y  for any 0 ≤ i ≤ j – 1 }}
  while (j !== S.length && S[j] !== y) {
    {{ (S[i] ≠ y  for any 0 ≤ i ≤ j – 1) and j ≠ len(S) and S[j] ≠ y }}
    {{ S[i] ≠ y  for any 0 ≤ i ≤ j }}
    j = j + 1;
    {{ S[i] ≠ y  for any 0 ≤ i ≤ j – 1 }}
  }
  return j !== S.length;
};
```

Is this valid?

Linear Search of an Array

$\{\{$ (S[i] $\neq$ y for any $0 \leq i \leq j - 1$) and $j \neq \text{len}(S)$ and S[j] $\neq$ y $\}\}$

$\{\{$ S[i] $\neq$ y for any $0 \leq i \leq j$ $\}\}$

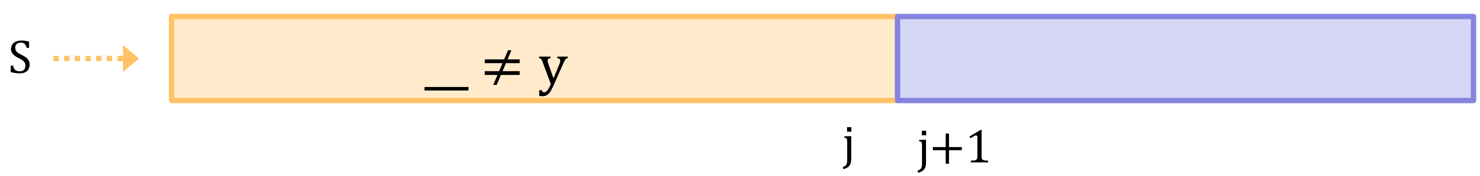
- **What does the top assertion say about S[j]?**
  - **it is not y**

Linear Search of an Array

$\{\{\ (S[i] \neq y \ \text{ for any } 0 \leq i \leq j - 1) \text{ and } j \neq \text{len}(S) \text{ and } S[j] \neq y\ \}\}$

$\{\{\ S[i] \neq y \ \text{ for any } 0 \leq i \leq j\ \}\}$

- **What is the picture for the bottom assertion?**

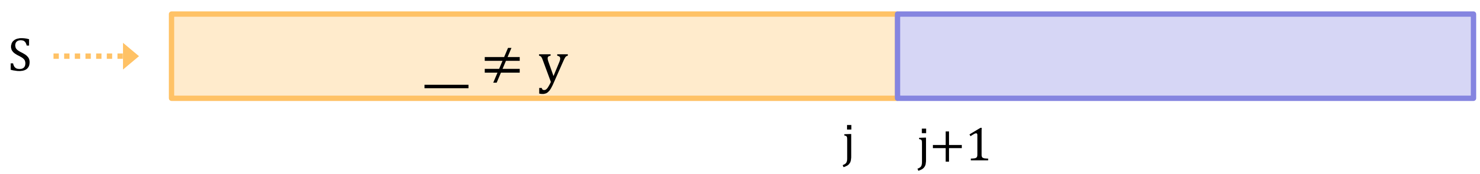


- **Do the facts above imply this holds?**
  - **Yes! It's the same picture**

Linear Search of an Array

{{ (S[i] $\neq$ y for any $0 \leq i \leq j - 1$) and j $\neq$ len(S) and S[j] $\neq$ y }}
{{ S[i] $\neq$ y for any $0 \leq i \leq j$ }}

- **What is the picture for the bottom assertion?**



- **Most likely bug is an off-by-one error**
  - **must check** S[j], **not** S[j-1] **or** S[j+1]

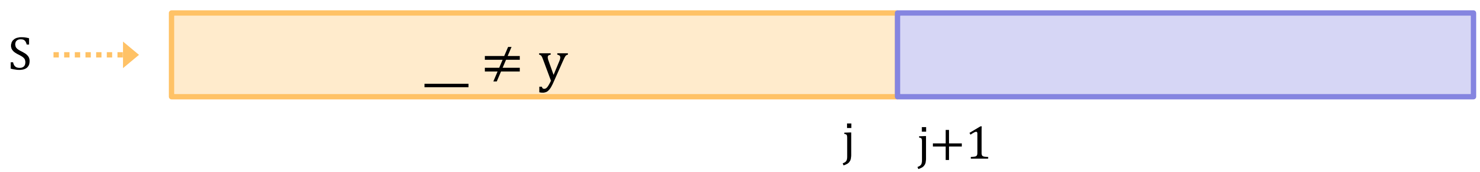
Linear Search of an Array


```
while (j !== S.length && S[j+1] !== y) {
```

{{ (S[i] $\neq$ y for any $0 \leq i \leq j - 1$) and j $\neq$ len(S) and S[j+1] $\neq$ y }}

{{ S[i] $\neq$ y for any $0 \leq i \leq j$ }}

- **What is the picture for the bottom assertion?**



- **Reasoning would verify that this is not correct**

Linear Search of an Array


```
const contains =
    (S: Array<bigint>, y: bigint): boolean => {
  let j = 0;
  {{ Inv: S[i] ≠ y  for any 0 ≤ i ≤ j – 1 }}
  while (j !== S.length && S[j] !== y) {
    j = j + 1;
  }
  {{ Inv and (j = len(S) or S[j] = y) }}
  {{ contains(S, y) = (j ≠ len(S)) }}
  return j !== S.length;
};
```

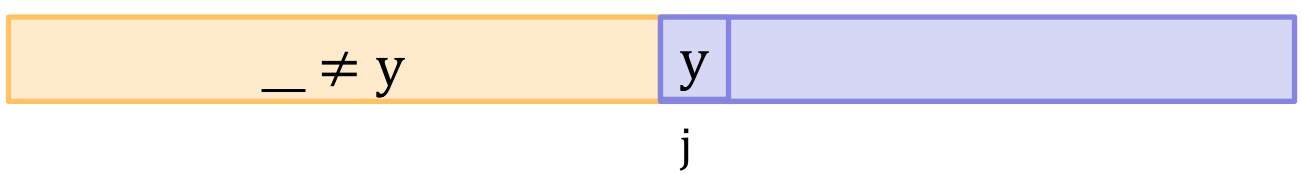
"or" means cases...

Case $j \neq \text{len}(S)$:

Must have $S[j] = y$.

What is the picture now?

Code should and does return true.



Linear Search of an Array


```
const contains =
    (S: Array<bigint>, y: bigint): boolean => {
  let j = 0;
  {{ Inv: S[i] ≠ y  for any 0 ≤ i ≤ j – 1 }}
  while (j !== S.length && S[j] !== y) {
    j = j + 1;
  }
  {{ Inv and (j = len(S) or S[j] = y) }}
  {{ contains(S, y) = (j ≠ len(S)) }}
  return j !== S.length;
};
```

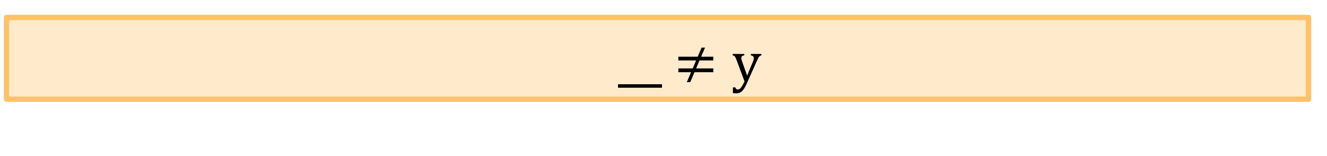
"or" means cases...

Case j = len(S):

What does Inv say now?

Says y is not in the array!

Code should and does return false.



Finding an Element in an Array

- Can search for an element in an array as follows

$$
\begin{aligned}
&\mathrm{contains}(\mathrm{nil}, y) && := \mathrm{false} && \\
&\mathrm{contains}(x :: L, y) && := \mathrm{true} && \textbf{if } x = y \\
&\mathrm{contains}(x :: L, y) && := \mathrm{contains}(L, y) && \textbf{if } x \neq y
\end{aligned}
$$

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

- Can be done more quickly if the list is sorted
  - binary search!

Finding an Element in a Sorted Array

- **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**

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

- **Not easy to describe this visually…**
  - **how about a gradient?**

Binary Search of an Array


```
const bsearch = (S: …, y: …): boolean => {
  let j = 0, k = S.length;
  {{ Inv: (S[i] < y  for any 0 ≤ i < j) and (y ≤ S[i] for any k ≤ i < n) }}
  while (j !== k) {
    const m = (j + k) / 2n;
    if (S[m] < y) {
      j = m + 1;
    } else {
      k = m;
    }
  }
  return (S[k] === y);
};
```

Inv includes facts about two regions.

Let's check that this is right...

Binary Search of an Array


```
const bsearch = (S: …, y: …): boolean => {
  let j = 0, k = S.length;
  {{ j = 0 and k = n }}
  {{ Inv: (S[i] < y  for any 0 ≤ i < j) and (y ≤ S[i] for any k ≤ i < n) }}
```

- **What does the picture look like with $j = 0$ and $k = n$?**

- **Does this hold?**
  - Yes! It's vacuously true

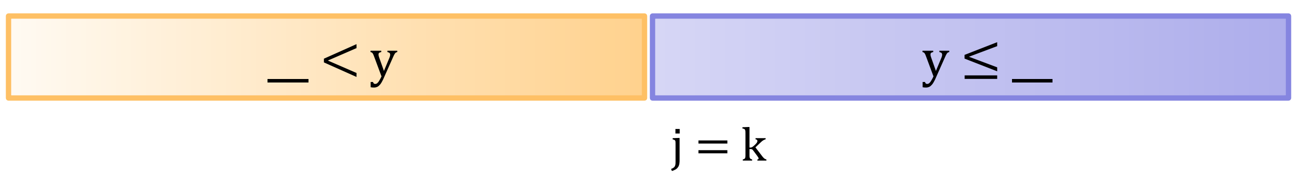
Binary Search of an Array


```
const bsearch = (S: …, y: …): boolean => {
  let j = 0, k = S.length;
  {{ Inv: (S[i] < y  for any 0 ≤ i < j) and (y ≤ S[i] for any k ≤ i < n) }}
  while (j !== k) {
   …
  }
  {{ Inv and (j = k) }}
  {{ contains(S, y) = (S[y] = y) }}
  return (S[k] === y);
};
```

Binary Search of an Array


```
    {{ Inv and (j = k) }}
    {{ contains(S, y) = (S[y] = y) }}
    return (S[k] === y);
};
```

- **What does the picture look like with $j = k$?**



- **Does S contain y iff $S[k] = y$?**
  - If $S[k] = y$, then $\text{contains}(S, y) = \text{true}$
  - If $S[k] \neq y$, then $S[k] < y$ and $S[i] < y$ for every $k < i$, so $\text{contains}(S, y) = \text{false}$

What case are we missing?

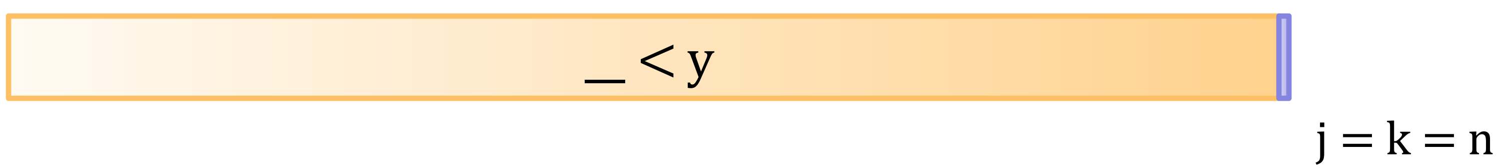
Binary Search of an Array

{{ Inv and $(j = k)$ }}

{{ $\text{contains}(S, y) = (S[y] = y)$ }}

```
return (S[k] === y);
};
```

- **What does the picture look like with $j = k = n$?**



- **In this case…**
  - **we see that** $\text{contains}(S, y) = \text{false}$
  - **and the code returns false because "**`undefined === y`**" is false**

    (Okay, but yuck.)

Binary Search of an Array


```
{{ Inv: (S[i] < y  for any 0 ≤ i < j) and (y ≤ S[i] for any k ≤ i < n) }}
while (j !== k) {
  {{ Inv and (j < k) }}
  const m = (j + k) / 2n;
  if (S[m] < y) {
    j = m + 1;
  } else {
    k = m;
  }
  {{ (S[i] < y  for any 0 ≤ i < j) and (y ≤ S[i] for any k ≤ i < n) }}
}
```

Reason through both paths...

Binary Search of an Array


```
{{ Inv and (j < k) }}
const m = (j + k) / 2n;
if (S[m] < y) {
  {{ Inv and (j < k) and (S[m] < y) }}
  j = m + 1;
} else {
  {{ Inv and (j < k) and (S[m] ≥ y) }}
  k = m;
}
{{ (S[i] < y  for any 0 ≤ i < j) and (y ≤ S[i] for any k ≤ i < n) }}
}
```

Binary Search of an Array


```
const m = (j + k) / 2n;
if (S[m] < y) {
```

{{ Inv and (j < k) and (S[m] < y) }}

{{ (S[i] < y for any 0 ≤ i < m+1) **and** (y ≤ S[i] for any k ≤ i < n) }}

```
  j = m + 1;
} else {
```

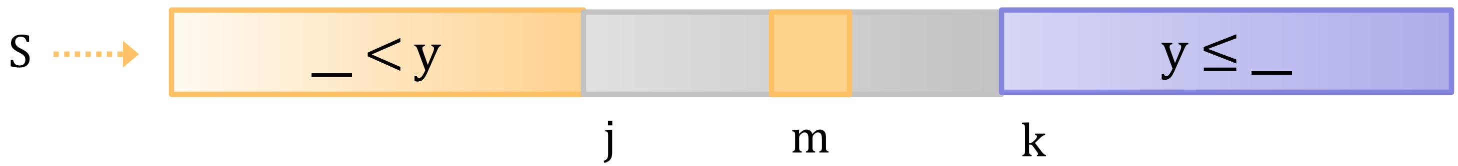
{{ Inv and (j < k) and (S[m] ≥ y) }}

{{ (S[i] < y for any 0 ≤ i < j) **and** (y ≤ S[i] for any m ≤ i < n) }}

```
  k = m;
}
```

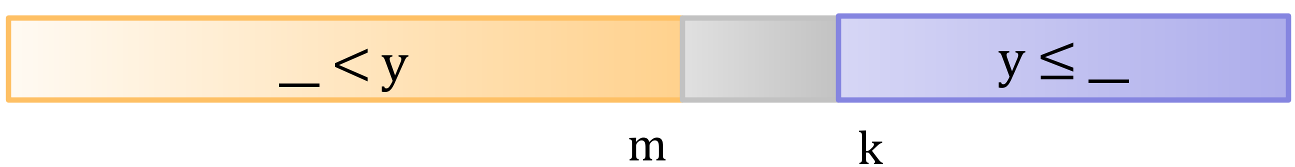
{{ (S[i] < y for any 0 ≤ i < j) **and** (y ≤ S[i] for any k ≤ i < n) }}

Binary Search of an Array



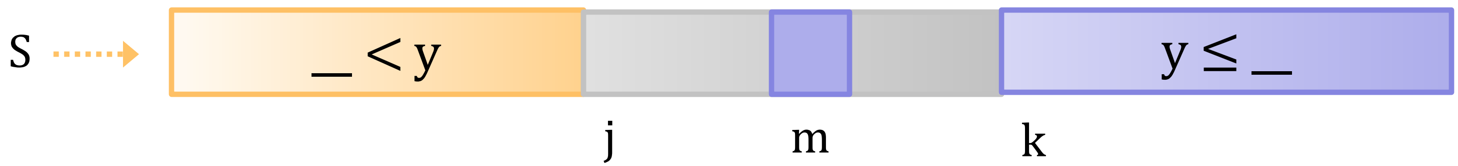

```
const m = (j + k) / 2n;
if (S[m] < y) {
  {{ Inv and (j < k) and (S[m] < y) }}
  {{ (S[i] < y  for any 0 ≤ i < m+1) and (y ≤ S[i] for any k ≤ i < n) }}
  j = m + 1;
} …
```

- **What does the picture look like in the bottom assertion?**



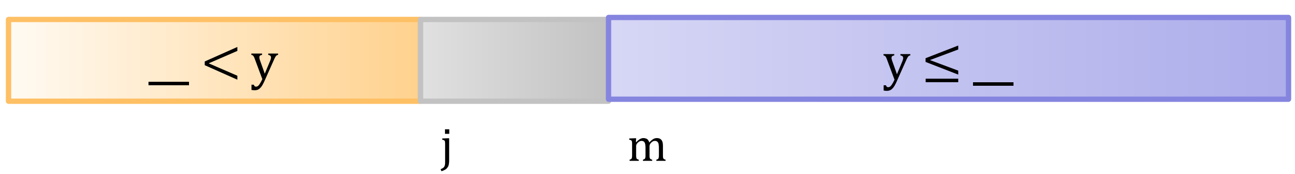
- **Does this hold?**
  - **Yes! Because the array is sorted (everything before S[m] is even smaller)**

Binary Search of an Array




```
const m = (j + k) / 2n;
… else {
  {{ Inv and (j < k) and (S[m] ≥ y) }}
  {{ (S[i] < y  for any 0 ≤ i < j) and (y ≤ S[i] for any m ≤ i < n) }}
  k = m;
}
```

- **What does the picture look like in the bottom assertion?**



- **Does this hold?**
  - **Yes! Because the array is sorted (everything after S[m] is even larger)**

Binary Search of an Array


```
const bsearch = (S: …, y: …): boolean => {
  let j = 0, k = S.length;
  {{ Inv: (S[i] < y  for any 0 ≤ i < j) and (y ≤ S[i] for any k ≤ i < n) }}
  while (j !== k) {
    const m = (j + k) / 2n;
    if (S[m] < y) {
      j = m + 1;
    } else {
      k = m;
    }
  }
  return (S[k] === y);
};
```

Does this terminate?

Need to check that $k - j$ *decreases*

Can see that $j \leq m \leq k$, so the "then" branch is fine.

Can see that $j < k$ implies $m < k$ (integer division rounds <u>down</u>), so the "else" branch is also fine

Loop Invariants

Loop Invariants with Arrays

- **Previous example:**

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

  {{ **Post**: $s = \text{sum}(S[0 .. n - 1])$ }}

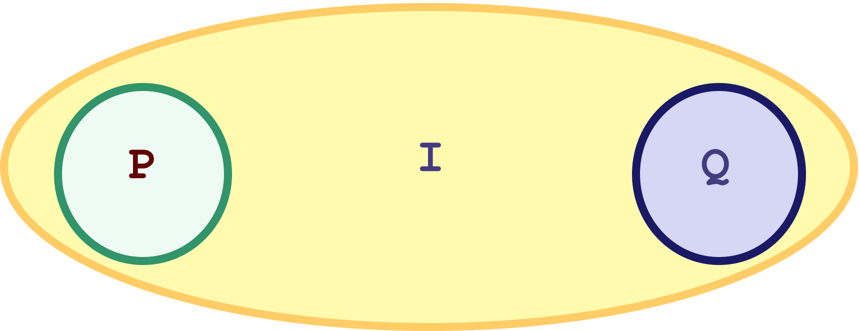
  - **in this case,** Post **is a special case of** Inv **(where** $j = n$**)**
  - **in other words,** Inv **is a weakening of** Post

- **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...

Loop Invariants with Arrays

- Previous example

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

  {{ **Post**: $s = \text{sum}(S[0 .. n-1])$ }}

- Linear search also fits this pattern:

  {{ **Inv**: $S[i] \neq y$ for any $0 \leq i < j$ }} **search an array**

  {{ **Post**: $(S[i] = y)$ or $(S[i] \neq y$ for any $0 \leq i < n)$ }}

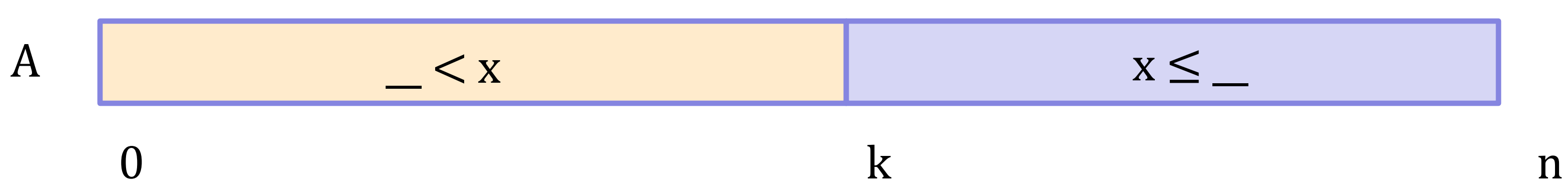
  – a **weakening** of second part

Searching a Sorted Array

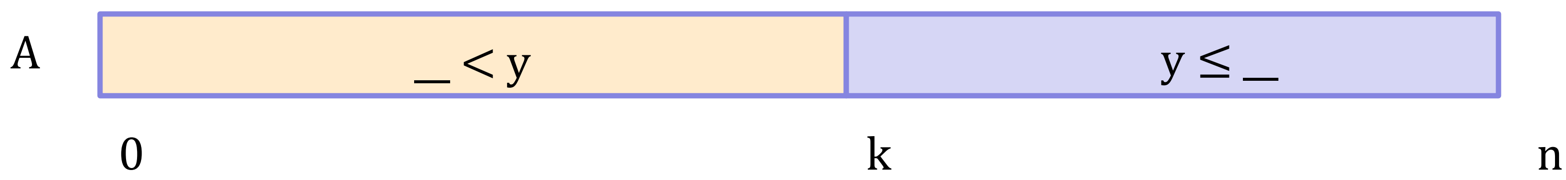
- **Suppose we require A to be sorted:**
  - **precondition includes**

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

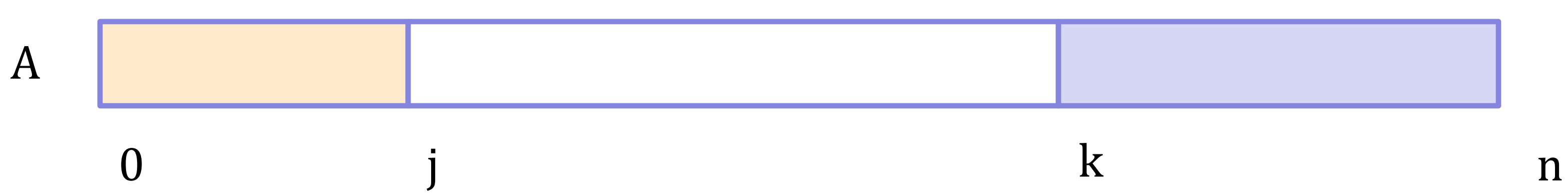
- **Want to find the index k where "x" would be...**
  - **picture would look like this:**



Searching a Sorted Array



- **End with complete knowledge of $A[i]$ vs x**
  - **how can we describe *partial* knowledge?**
  - **know some elements are smaller and some larger**



$A[i] < y$ for any $0 \le i < j$ $\quad$ $y \le A[i]$ for any $k \le i < n$

Loop Invariants with Arrays

- **Previous example**

{{ **Inv**: $s = \text{sum}(S[0 .. j-1]) \ldots$ }} **sum of array**

{{ **Post**: $s = \text{sum}(S[0 .. n-1])$ }}

- **Linear search also fits this pattern:**

{{ **Inv**: $S[i] \neq y$ for any $0 \leq i < j$ }} **search an array**

{{ **Post**: $(S[i] = y)$ or $(S[i] \neq y$ for any $0 \leq i < n)$ }}

- **Binary search also still fits this pattern**

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

{{ **Post**: $(S[i] < y$ for any $0 \leq i < \mathbf{k})$ **and** $(y \leq S[i]$ for any $k \leq i < n)$ }}

Loop Invariants

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

- 421 covers complex heuristics for finding invariants…
  - for 331, this heuristic is enough
  - (will give you the invariant for anything more complex)

Writing Loops

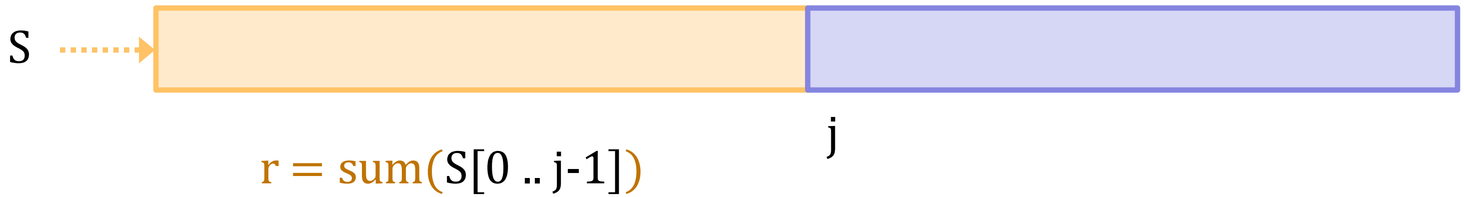
Writing Loops

- Examples so far have been <u>code reviews</u>
  - checking correctness of given code

- Steps to write a loop to solve a problem:
  1. Come up with an idea for the loop
  2. Formalize the idea in the invariant
  3. Write the code so that it is correct with that invariant

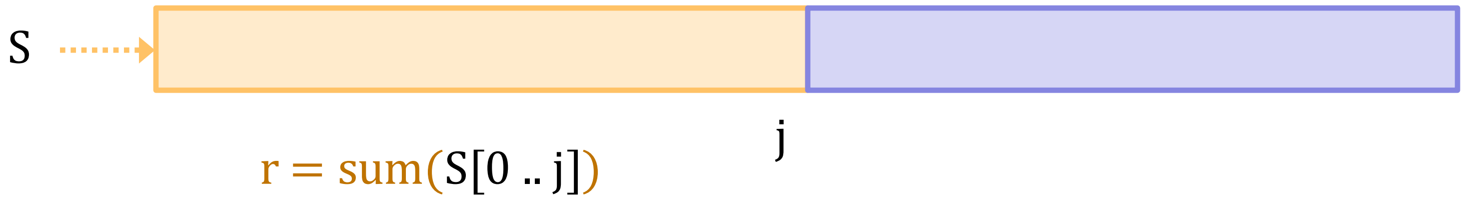
- Let's see some examples…

Recall: Sum of an Array




```
const sum = (S: Array<bigint>): bigint => {
  let r = 0;
  let j = 0;
  // Inv: r = sum(S[0 .. j-1])
  while (j != S.length) {
    r = S[j] + r;
    j = j + 1;
  }
  return r;
};
```

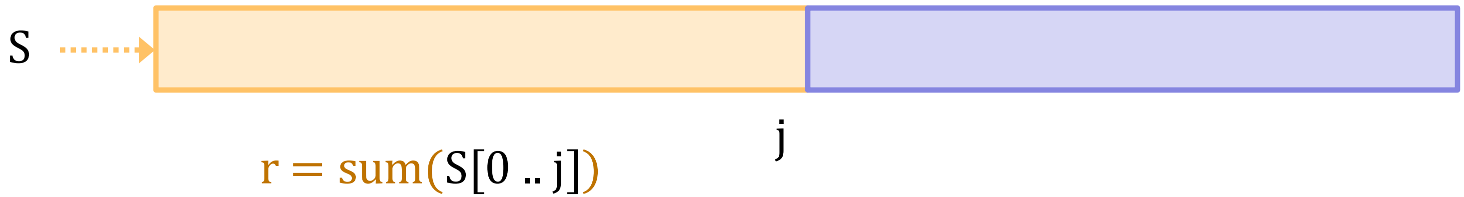
Sum of an Array (version 2)




```
const sum = (S: Array<bigint>): bigint => {
  let r = 0;
  let j = ??
  // Inv: r = sum(S[0 .. j])
  while (??) {
    r = ??
    j = j + 1;
  }
  return r;
};
```

How do we fill in the blanks to make this code correct?

Sum of an Array (version 2)

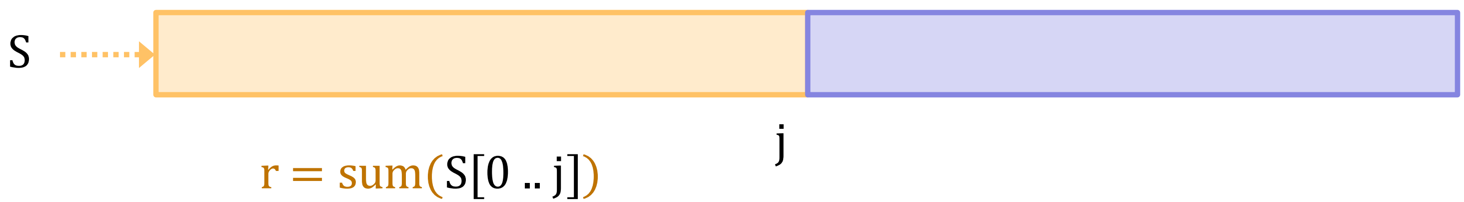



```
const sum = (S: Array<bigint>): bigint => {
  let r = 0;
  let j = ??
  // Inv: r = sum(S[0 .. j])
```

- **What do we set j to so that $\mathrm{sum}(S[0 .. j]) = 0$?**
  - **must set it to -1:**

$$\mathrm{sum}(S[0 .. -1]) = \mathrm{sum}([]) = 0$$

Sum of an Array (version 2)

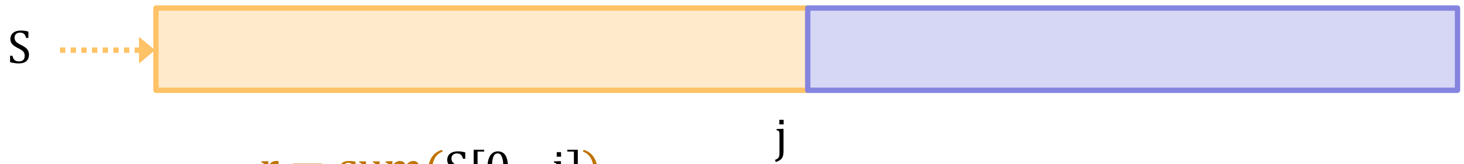



```
const sum = (S: Array<bigint>): bigint => {
  let r = 0;
  let j = -1;
  // Inv: r = sum(S[0 .. j])
  while (??) {
    ...
  }
  {{ Post: r = sum(S[0 .. n-1]) }}
  return r;
};
```

When do we exit to ensure that $\text{sum}([0 \,..\, j]) = \text{sum}(S[0 \,..\, n\text{-}1])$**?**

Exit when $j = n - 1$

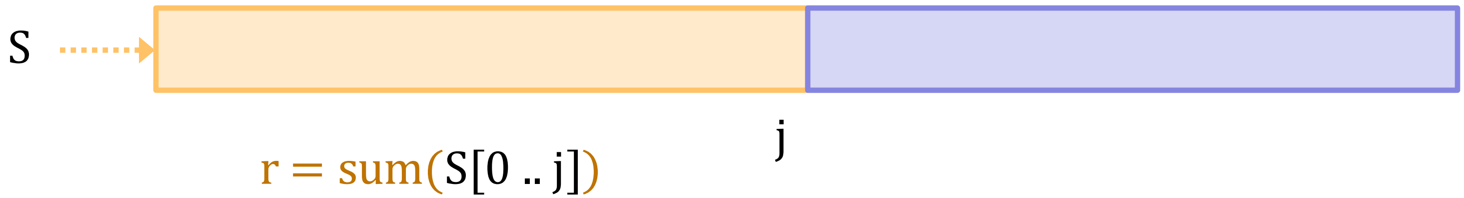
Sum of an Array (version 2)



r = sum(S[0 .. j])

```
const sum = (S: Array<bigint>): bigint => {
  let r = 0;
  let j = -1;
  // Inv: r = sum(S[0 .. j])
  while (j !== S.length - 1) {
    {{ r = sum(S[0 .. j]) and j ≠ n – 1 }}
    r = ??
    j = j + 1;
    {{ r = sum(S[0 .. j]) }}
  }
  return r;
};
```

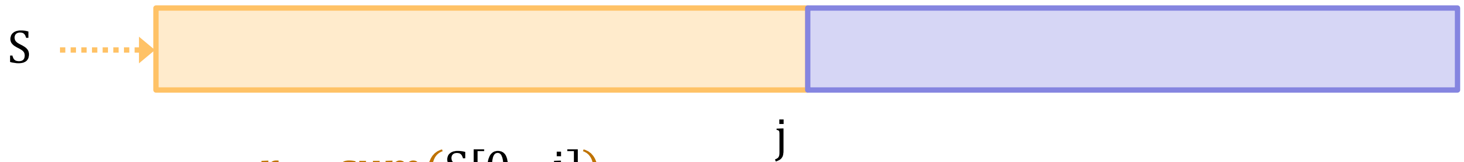
Sum of an Array (version 2)




```
const sum = (S: Array<bigint>): bigint => {
  let r = 0;
  let j = -1;
  // Inv: r = sum(S[0 .. j])
  while (j !== S.length - 1) {
    {{ r = sum(S[0 .. j]) and j ≠ n – 1 }}
    r = ??
    {{ r = sum(S[0 .. j+1]) }}
    j = j + 1;
    {{ r = sum(S[0 .. j]) }}
  }
```

Let's draw the second picture...

Sum of an Array (version 2)



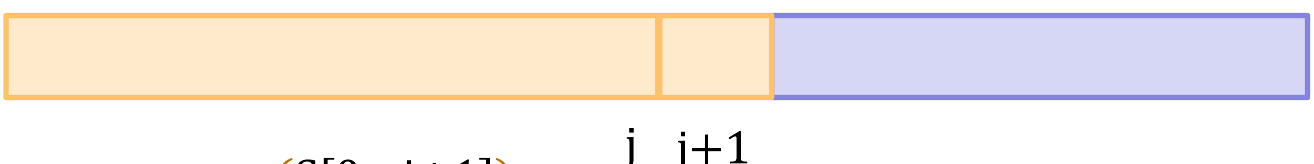
$r = \mathrm{sum}(S[0 .. j])$

$\{\{\ r = \mathrm{sum}(S[0 .. j]) \text{ and } j \neq n-1\ \}\}$

```
r = ??
```

$\{\{\ r = \mathrm{sum}(S[0 .. j+1])\ \}\}$

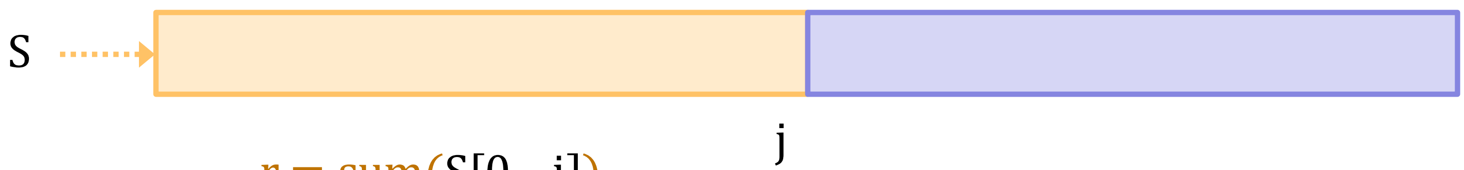
- **What is the picture in the second case?**



$r = \mathrm{sum}(S[0 .. j+1])$

- **What do we add to r to make this hold?**
  - **must add in** $S[j+1]$

Sum of an Array (version 2)



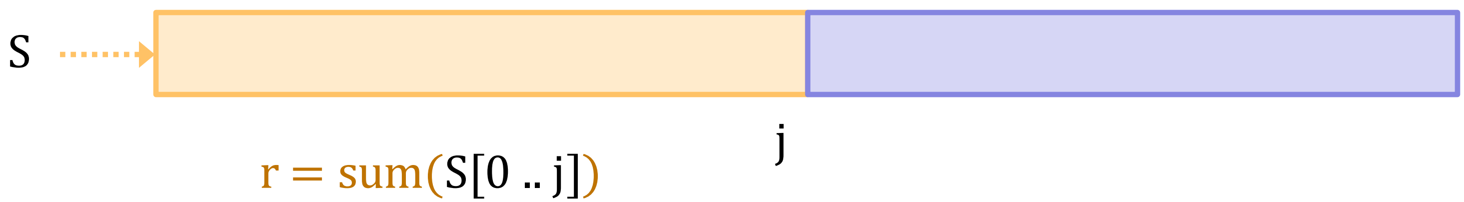
$r = \text{sum}(S[0 \,..\, j])$

```
const sum = (S: Array<bigint>): bigint => {
  let r = 0;
  let j = -1;
  // Inv: r = sum(S[0 .. j])
  while (j !== S.length - 1) {
    r = S[j+1] + r;
    j = j + 1;
  }
  return r;
};
```

This code is correct by construction.

Different from $r = \text{sum}(S[0 \,..\, j\text{-}1])$ **but does the same thing.**

Sum of an Array (version 3)

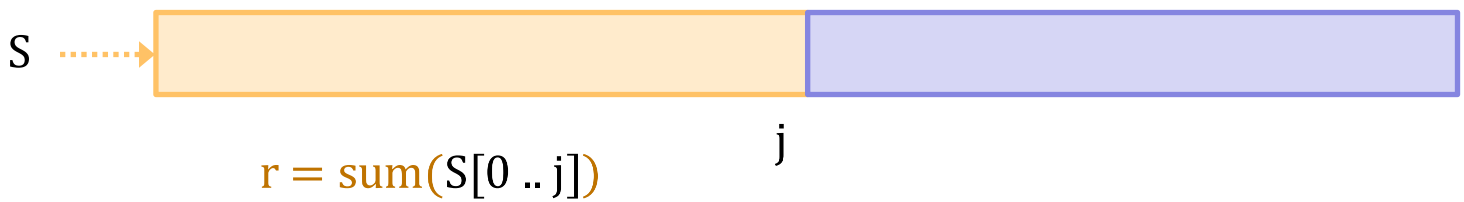



```
const sum = (S: Array<bigint>): bigint => {
  let r = 0;
  let j = -1;
  // Inv: r = sum(S[0 .. j])
  while (j !== S.length - 1) {
    j = j + 1;
    r = ??
  }
  return r;
};
```

What if we wrote it this way?

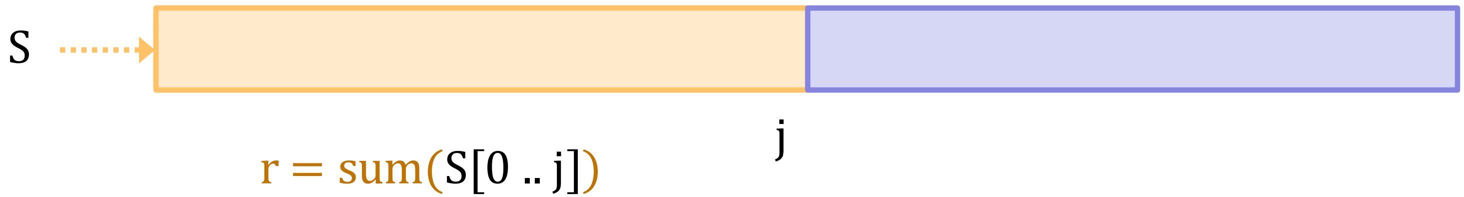
Same Inv but increase j at the start.

Sum of an Array (version 3)




```
const sum = (S: Array<bigint>): bigint => {
  let r = 0;
  let j = -1;
  // Inv: r = sum(S[0 .. j])
  while (j !== S.length - 1) {
    {{ r = sum(S[0 .. j]) and j ≠ n – 1 }}
    j = j + 1;
    r = ??
    {{ r = sum(S[0 .. j]) }}
  }
  return r;
};
```

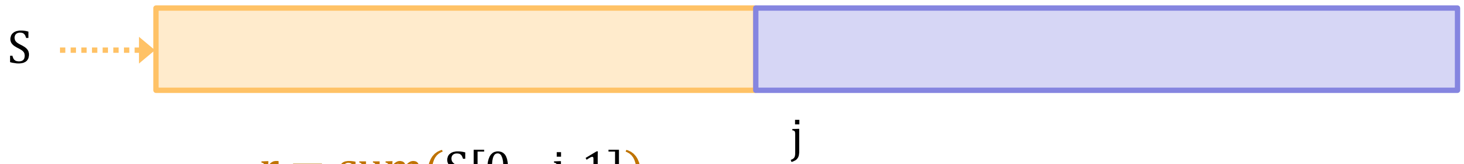
Sum of an Array (version 3)




```
const sum = (S: Array<bigint>): bigint => {
  let r = 0;
  let j = -1;
  // Inv: r = sum(S[0 .. j])
  while (j !== S.length - 1) {
    {{ r = sum(S[0 .. j]) and j ≠ n – 1 }}
    j = j + 1;
    {{ r = sum(S[0 .. j-1]) and j-1 ≠ n – 1 }}
    r = ??
    {{ r = sum(S[0 .. j]) }}
  }
```

Let's draw these pictures...

Sum of an Array (version 3)

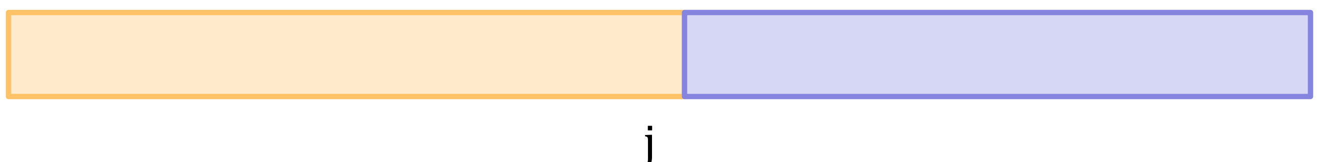


$r = \mathrm{sum}(S[0\ ..\ j\text{-}1])$

$\{\{\ r = \mathrm{sum}(S[0\ ..\ j\text{-}1])\ \text{and}\ j - 1 \neq n - 1\ \}\}$

```
r = ??
```

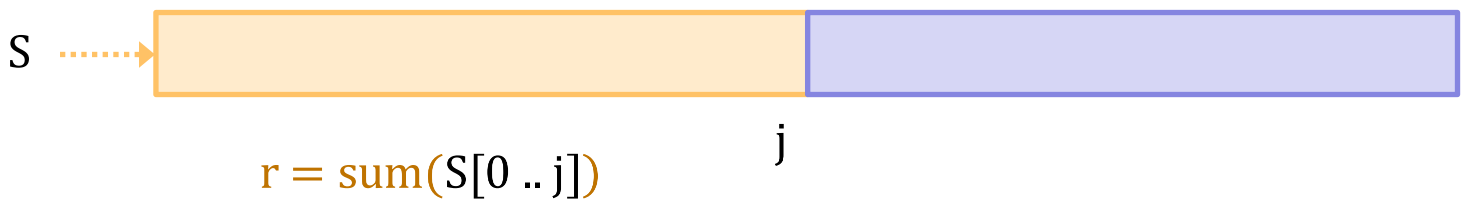
$\{\{\ r = \mathrm{sum}(S[0\ ..\ j])\ \}\}$



$r = \mathrm{sum}(S[0\ ..\ j])$

- **What do we add to r to make this hold?**
  - **must add in** $S[j]$

Sum of an Array (version 3)

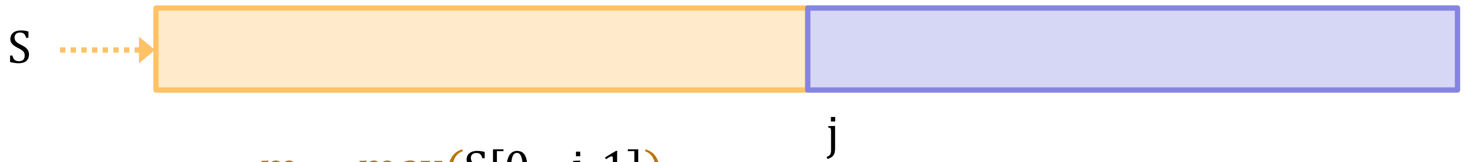



```
const sum = (S: Array<bigint>): bigint => {
  let r = 0;
  let j = -1;
  // Inv: r = sum(S[0 .. j])
  while (j !== S.length - 1) {
    j = j + 1;
    r = S[j] + r;
  }
  return r;
};
```

Changing Inv or `j = …` line (loop idea) changes the code we need to write.

Once the loop idea is formalized, can fill in the code to make it correct.

Max of an Array



$m = \max(S[0 .. j\text{-}1])$

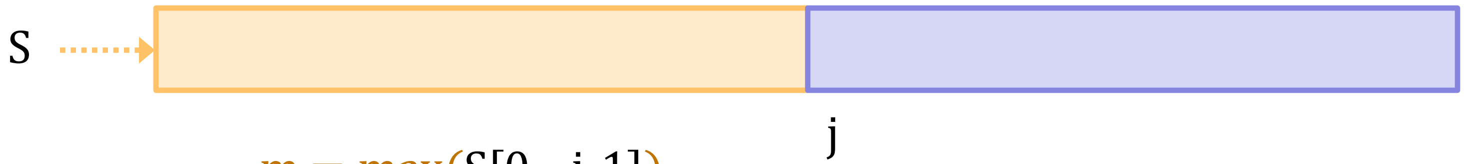
```
const max = (S: Array<bigint>): bigint => {
  let m = ??
  let j = ??
  // Inv: m = max(S[0 .. j-1])
  while (??) {
    ??
    j = j + 1;
  }
  return m;
};
```

How do we initialize m & j?

$m = \max(S[0 .. 0])$ **is easiest**

What case is missing?

Max of an Array



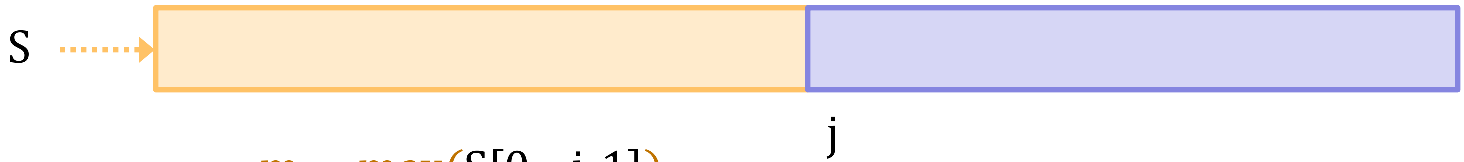
m = max(S[0 .. j-1])

```
const max = (S: Array<bigint>): bigint => {
  if (S.length === 0) throw new Error('no elements);
  let m = S[0];
  let j = ??
  // Inv: m = max(S[0 .. j-1])
  while (??) {
    ??
    j = j + 1;
  }
  return m;
};
```

How do we initialize j?

Want m = max(S[0 .. 0])

Max of an Array



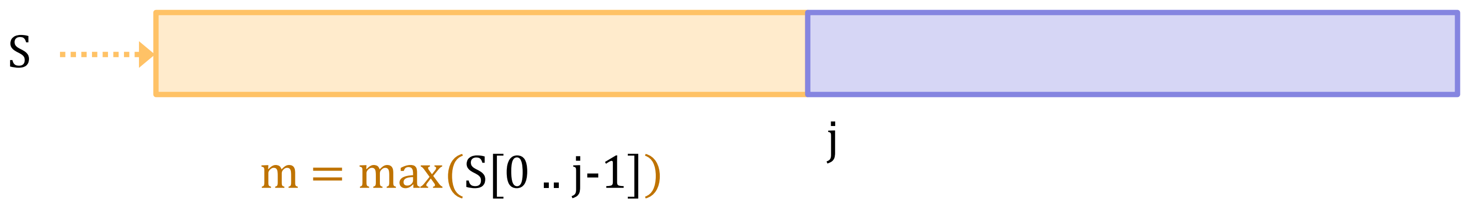
m = max(S[0 .. j-1])

```
const max = (S: Array<bigint>): bigint => {
  if (S.length === 0) throw new Error('no elements);
  let m = S[0];
  let j = 1;
  // Inv: m = max(S[0 .. j-1])
  while (??) {
    ??
    j = j + 1;
  }
  return m;
};
```

When do we exit?

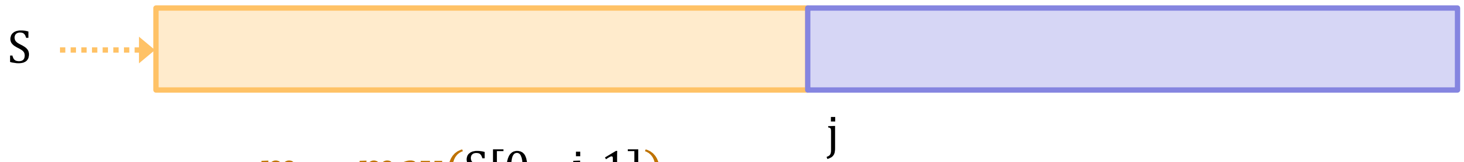
Want m = max(S[0 .. n-1])

Max of an Array




```
const max = (S: Array<bigint>): bigint => {
  if (S.length === 0) throw new Error('no elements);
  let m = S[0];
  let j = 1;
  // Inv: m = max(S[0 .. j-1])
  while (j !== S.length) {
    ??
    j = j + 1;
  }
  return m;
};
```

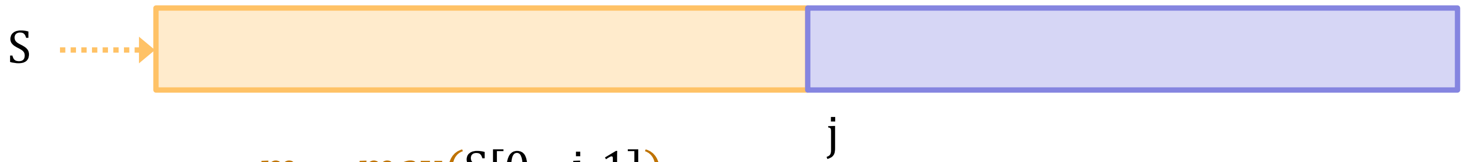
Max of an Array



m = max(S[0 .. j-1])

```
const max = (S: Array<bigint>): bigint => {
  if (S.length === 0) throw new Error('no elements);
  let m = S[0];
  let j = 1;
  // Inv: m = max(S[0 .. j-1])
  while (j !== S.length) {
    {{ m = max(S[0 .. j-1]) and j ≠ n }}
    ??
    {{ m = max(S[0 .. j]) }}
    j = j + 1;
  }
```

Max of an Array

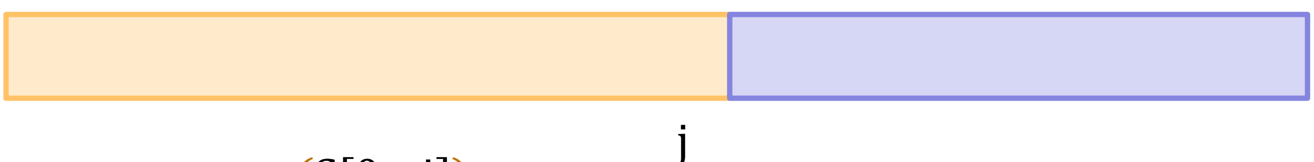


$m = \max(S[0 .. j\text{-}1])$

$\{\{\ m = \max(S[0 .. j\text{-}1]) \text{ and } j \neq n\ \}\}$

??

$\{\{\ m = \max(S[0 .. j])\ \}\}$

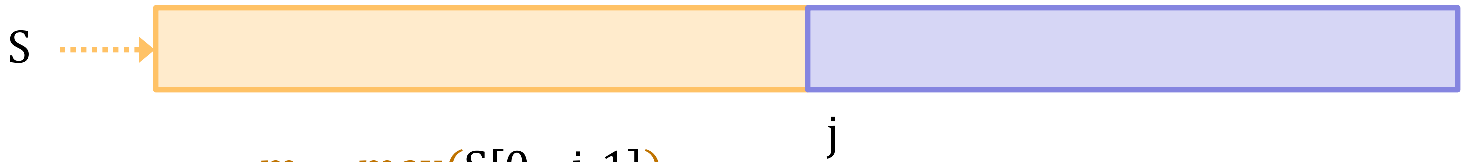


$m = \max(S[0 .. j])$

How do we make the second one hold?

Set $m = S[j]$ **iff** $S[j] > m$

Max of an Array



m = max(S[0 .. j-1])

```
const max = (S: Array<bigint>): bigint => {
  if (S.length === 0) throw new Error('no elements);
  let m = S[0];
  let j = 1;
  // Inv: m = max(S[0 .. j-1])
  while (j !== S.length) {
    if (S[j] > m)
      m = S[j];
    j = j + 1;
  }
  return m;
};
```

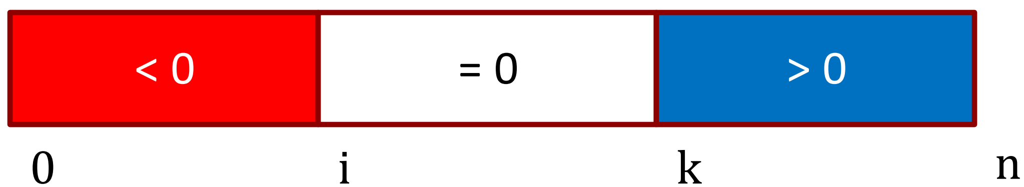
Example: Sorting Negative, Zero, Positive

- 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?

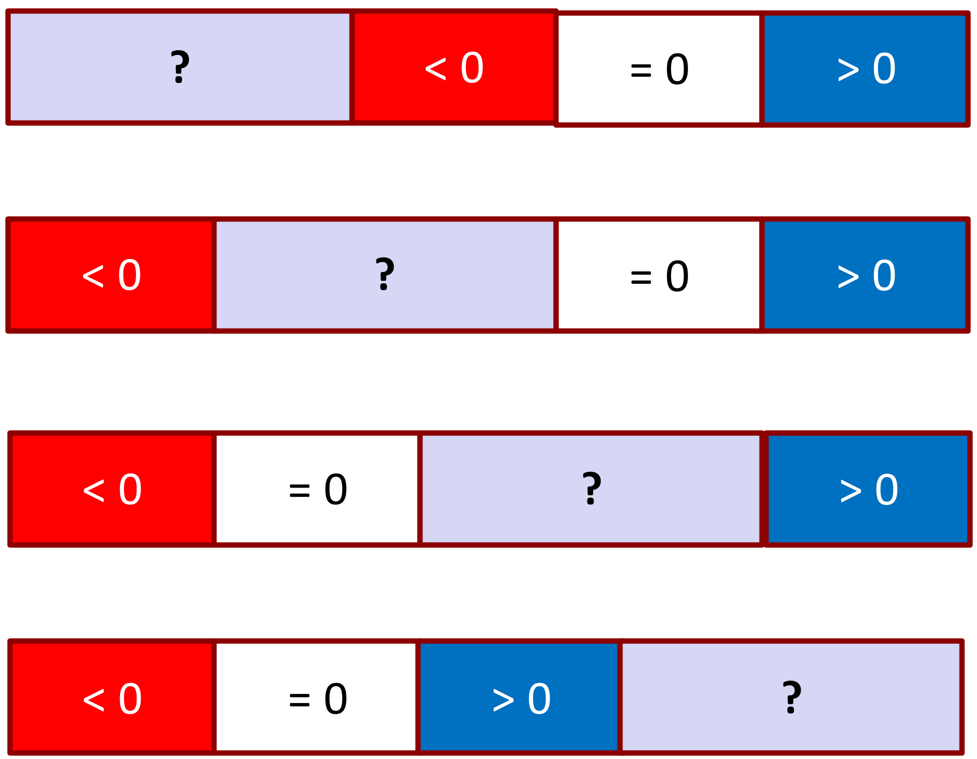
– weaken the postcondition

Example: Sorting Negative, Zero, Positive

How should we weaken this for the invariant?

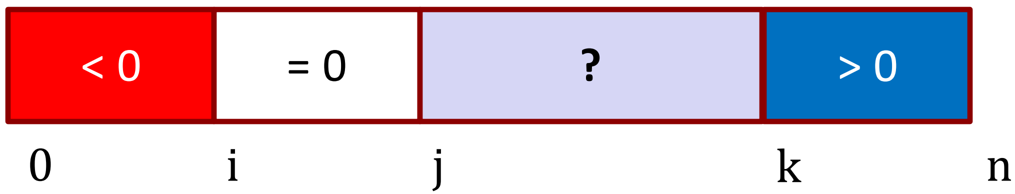
- **needs allow elements with *unknown* values**

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



Example: Sorting Negative, Zero, Positive

Our Invariant:



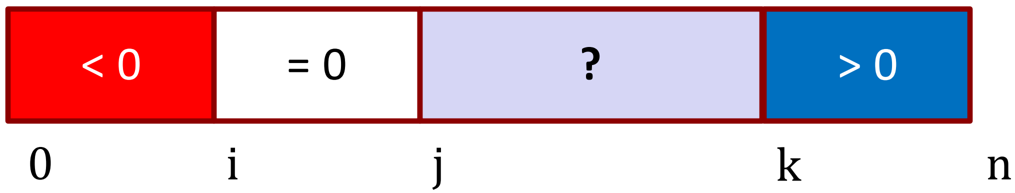
$A[\ell] < 0$ for any $0 \leq \ell < i$

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

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

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

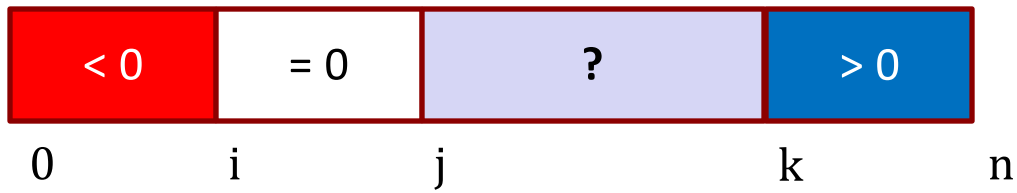
Example: Sorting Negative, Zero, Positive



- Let's try figuring out the code to make it correct

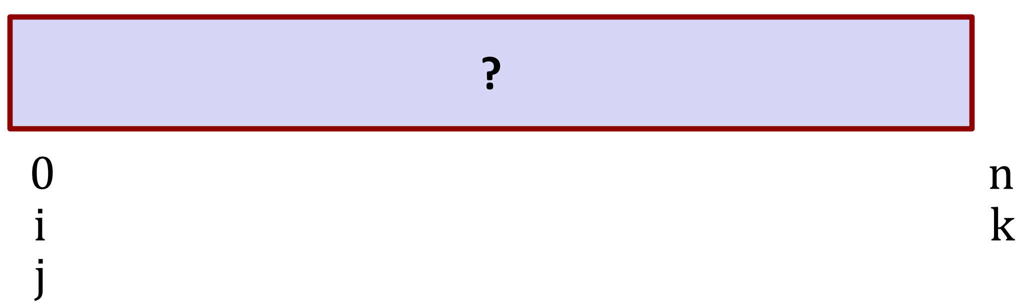
- Figure out the code for
  - how to initialize
  - when to exit
  - loop body

Example: Sorting Negative, Zero, Positive

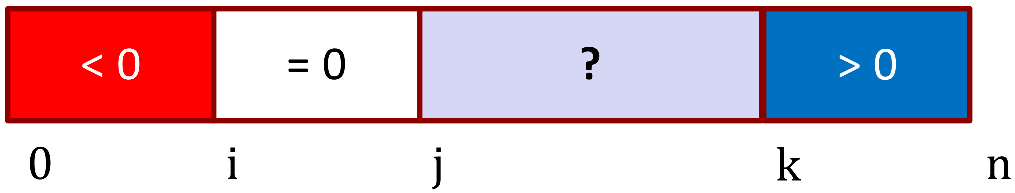


- **Will have variables** i, j, **and** k **with** $i \leq j \leq k$

- **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 $k = n$

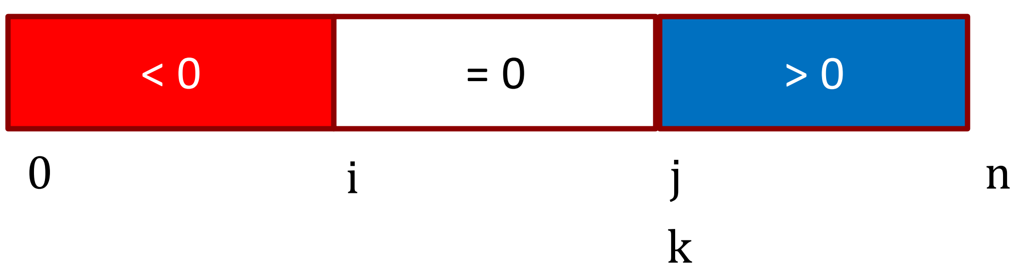


Example: Sorting Negative, Zero, Positive



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

- **When do we exit?**
  - purple is empty if $j = k$



Sort Positive, Zero, Negative

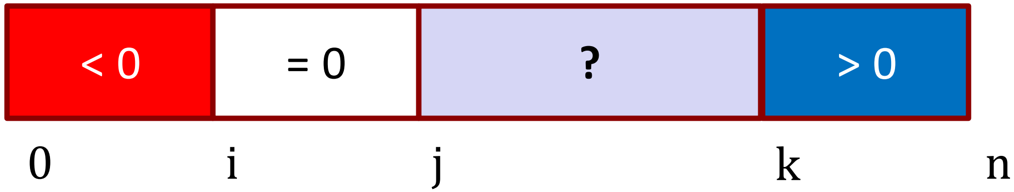
```
let i = 0;
let j = 0;
let k = A.length;
```

{{ **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 $0 \le i \le j \le k \le n$}}

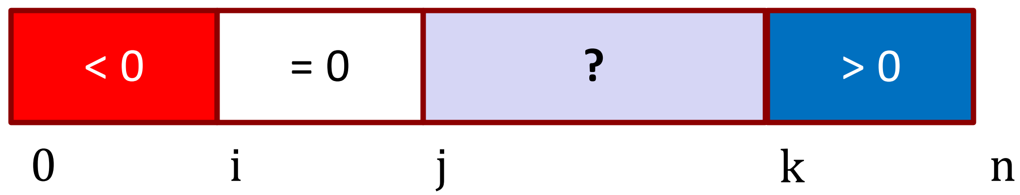
```
while (j < k) {
  ...
}
```

{{ $A[\ell] < 0$ 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

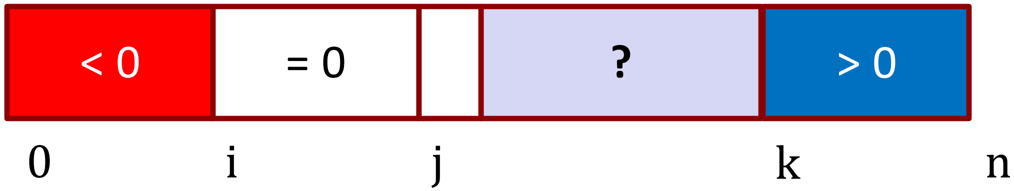


- **How do we make progress?**
  - try to increase j by 1 or decrease k by 1

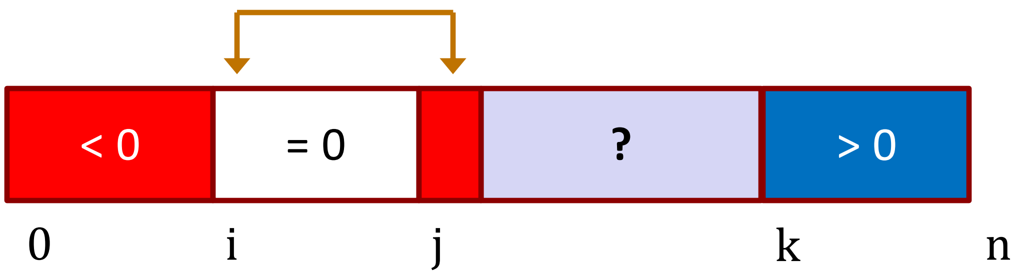
- **Look at $A[j]$ and figure out where it goes**

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

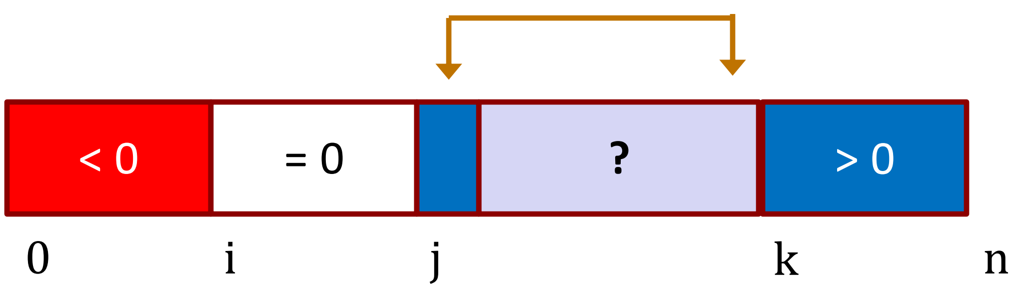
Example: Sorting Negative, Zero, Positive



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 $k = k_0 - 1$

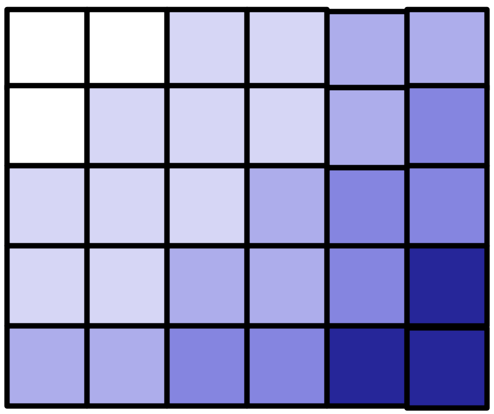
Sort Positive, Zero, Negative

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

Sorted Matrix Search

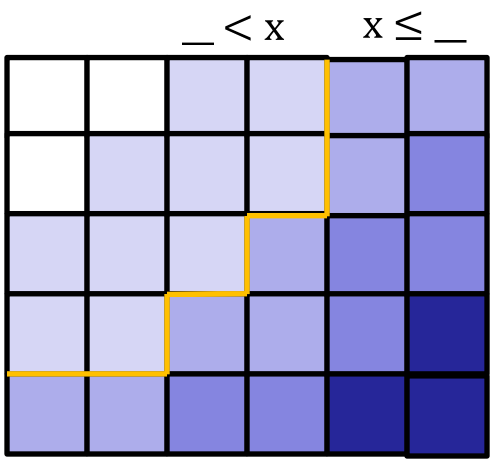
Given a sorted matrix M, with m rows and n cols, where every row and every column is sorted, find out whether a given number x is in the matrix



(darker color means larger)

Sorted Matrix Search

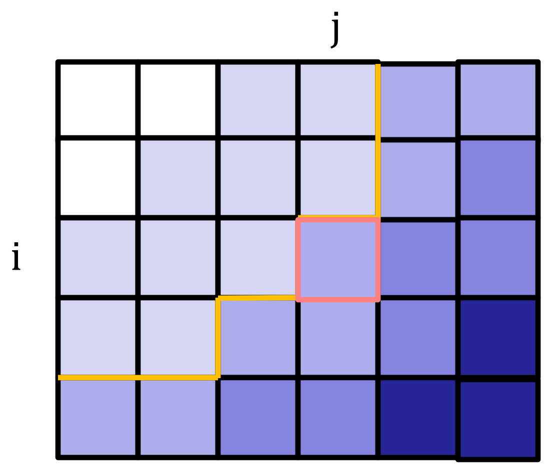
Given a sorted matrix M, with m rows and n cols, where every row and every column is sorted, find out whether a given number x is in the matrix



Idea: Trace the contour between the numbers ≤ x and > x in each row to see if x appears.

Sorted Matrix Search

Given a sorted matrix M, with m rows and n cols, where every row and every column is sorted, find out whether a given number x is in the matrix



Invariant: at the left-most entry with $x \leq$ _ in the row

- for each row i, this holds for exactly one column j

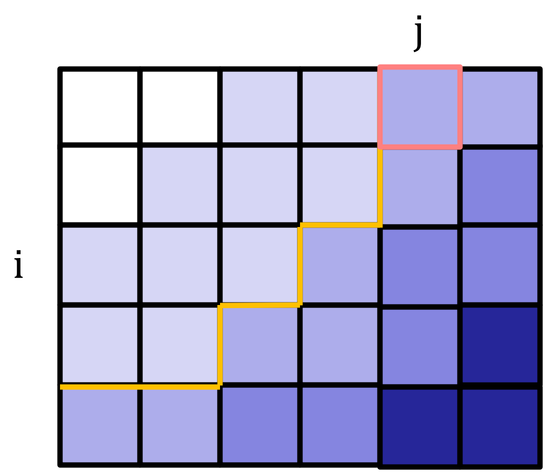
Sorted Matrix Search

Invariant: at the left-most entry with $x \leq$ _ in the row

– for each row i, this holds for exactly one column j

Initialization: how do we get this to hold for $i = 0$?

– could be anywhere in the first row

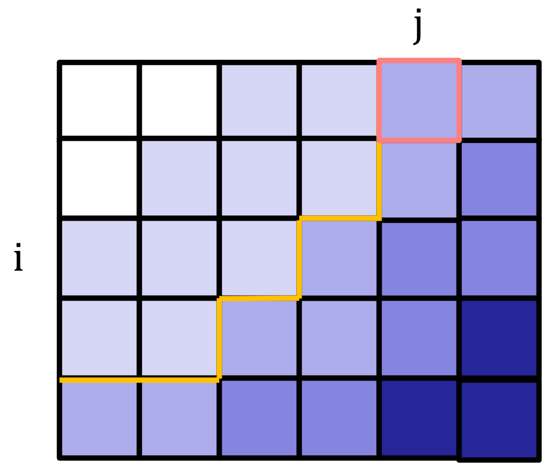


Need to *search* to find this location

Sorted Matrix Search

New Goal: find smallest j with $x \leq M[0, \mathbf{k}]$ for any $j \leq k < n$

– will need a loop…



How do we find an invariant for that loop?

– try weakening this assertion (allow any j, not just smallest)

– decrease j until $x \leq M[0, j\text{-}1]$ does not hold

Sorted Matrix Search

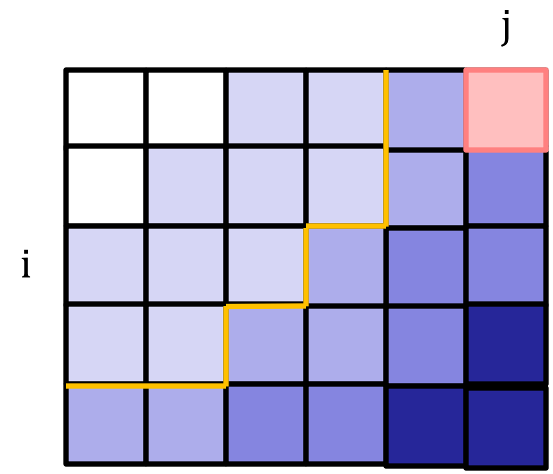
New Goal: find smallest j with $x \leq M[0, \mathbf{k}]$ for any $j \leq k < n$

```
let i = 0;
let j = ??
```

{{ **Inv**: $x \leq M[0, k]$ for any $j \leq k < n$ }}

```
while (??)
  ??
```

{{ **Post**: $M[0, k] < x$ for any $0 \leq k < j$ and $x \leq M[0, k]$ for any $j \leq k < n$ }}

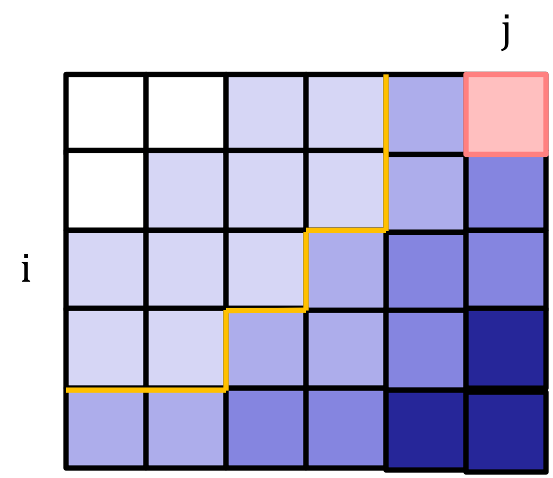


How do we set j to make Inv hold initially?

– range is empty when $j = n$

Sorted Matrix Search

New Goal: find smallest j with $x \leq M[0, \mathbf{k}]$ for any $j \leq k < n$




```
let i = 0;
let j = n;
```

{{ **Inv**: $x \leq M[0, k]$ for any $j \leq k < n$ }}

```
while (??)
   ??
```

{{ **Post**: $M[0, k] < x$ for any $0 \leq k < j$ and $x \leq M[0, k]$ for any $j \leq k < n$ }}

How do we exit so that the postcondition holds?

- can no longer decrease j when $j = 0$ or $M[0, j\text{-}1] < x$

Sorted Matrix Search

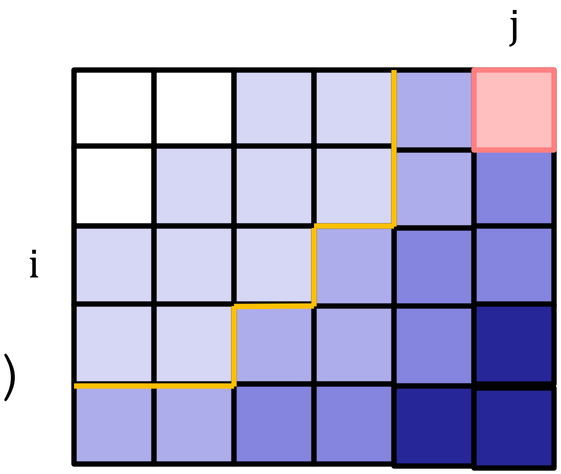
New Goal: find smallest j with $x \le M[0, \mathbf{k}]$ for any $j \le k < n$

```
let i = 0;
let j = n;
```

{{ **Inv**: $x \le M[0, k]$ for any $j \le k < n$ }}

```
while (j>0 && x <= M[0][j-1])
   ??
```

{{ **Post**: $M[0, k] < x$ for any $0 \le k < j$ and $x \le M[0, k]$ for any $j \le k < n$ }}



Anything needed in the loop body?
(That is, other than `j = j - 1`?)

Sorted Matrix Search

New Goal: find smallest j with $x \le M[0, \mathbf{k}]$ for any $j \le k < n$

```
{{ Inv: x ≤ M[0, k] for any j ≤ k < n }}
while (j>0 && x <= M[0][j-1]) {
    {{ x ≤ M[0, k] for any j ≤ k < n and j > 0 and x ≤ M[0, j-1] }}
    ??
    j = j - 1;
    {{ x ≤ M[0, k] for any j ≤ k < n }}
}
```

Sorted Matrix Search

New Goal: find smallest j with $x \leq M[0, \mathbf{k}]$ for any $j \leq k < n$

{{ **Inv**: $x \leq M[0, k]$ for any $j \leq k < n$ }}

```
while (j>0 && x <= M[0][j-1]) {
```

{{ $x \leq M[0, k]$ for any $j \leq k < n$ and $j > 0$ and $x \leq M[0, j\text{-}1]$ }}

??

{{ $x \leq M[0, k]$ for any $j - 1 \leq k < n$ }}

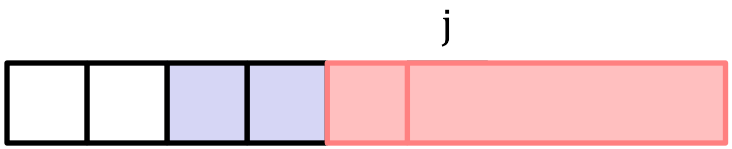
```
  j = j - 1;
```

{{ $x \leq M[0, k]$ for any $j \leq k < n$ }}

```
}
```

Sorted Matrix Search

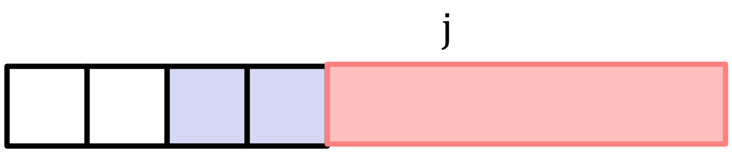
New Goal: **find smallest** j **with** $x \leq M[0, \mathbf{k}]$ for any $j \leq k < n$



{{ $x \leq M[0, k]$ for any $j \leq k < n$ and $j > 0$ and $x \leq M[0, j\text{-}1]$ }}

??

{{ $x \leq M[0, k]$ for any $j - 1 \leq k < n$ }}



Nothing is missing!

Sorted Matrix Search

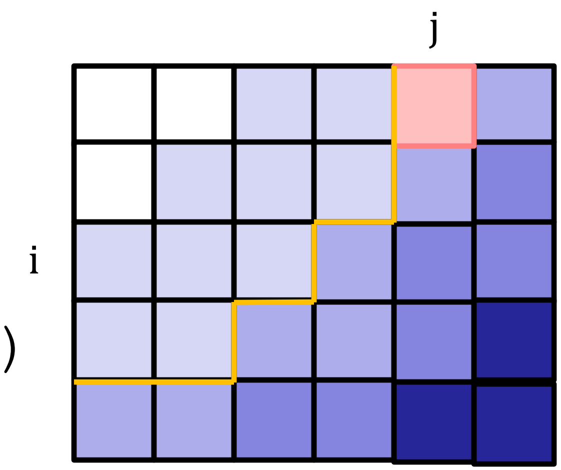
New Goal: find smallest j with $x \leq M[0, \mathbf{k}]$ for any $j \leq k < n$

```
let i = 0;
let j = n;
```

{{ **Inv**: $x \leq M[0, k]$ for any $j \leq k < n$ }}

```
while (j>0 && x <= M[0][j-1])
  j = j - 1;
```

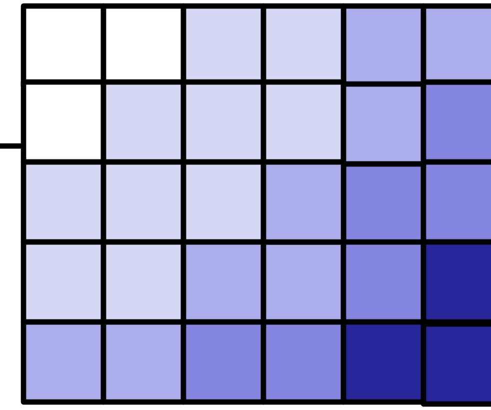
{{ **Post**: $M[0, k] < x$ for any $0 \leq k < j$ and $x \leq M[0, k]$ for any $j \leq k < n$ }}



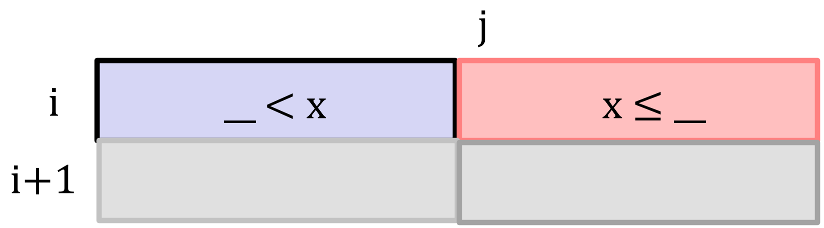
Can now check if $M[0, j] = x$

- if not, then it is not in the first row
- move on to the second row...

Sorted Matrix Search



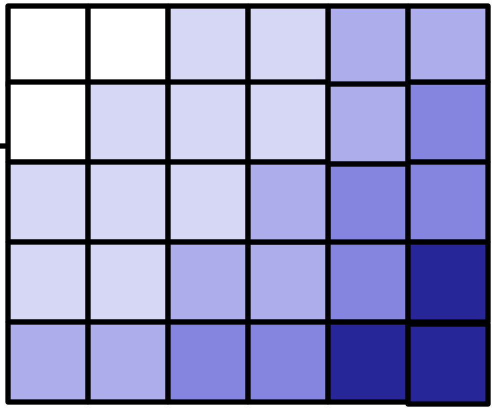
Moving from row i to row i+1



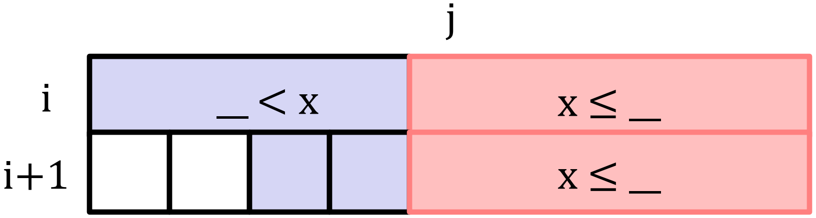
What does *vertical* sorting tell us about row $i+1$?

- right side is guaranteed to satisfy " $x \leq \_$ "
- left side **not** guaranteed to satisfy " $\_ < x$ "

Sorted Matrix Search

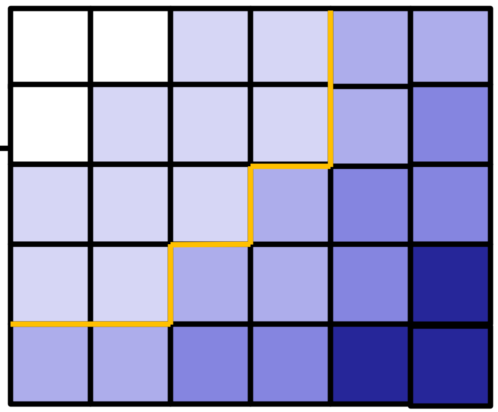


Moving from row i to row i+1

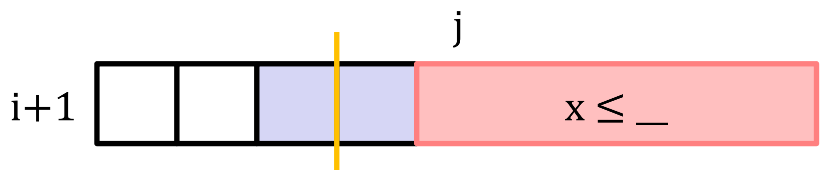


Next row looks like this

Sorted Matrix Search



Moving from row i to row i+1



How do we restore the invariant?

– find the index j with $M[i+1, j\text{-}1] < x \leq M[i+1, j]$

This is the same problem as before!

– move left until begining or $M[i+1, j\text{-}1] < x$ holds

Sorted Matrix Search

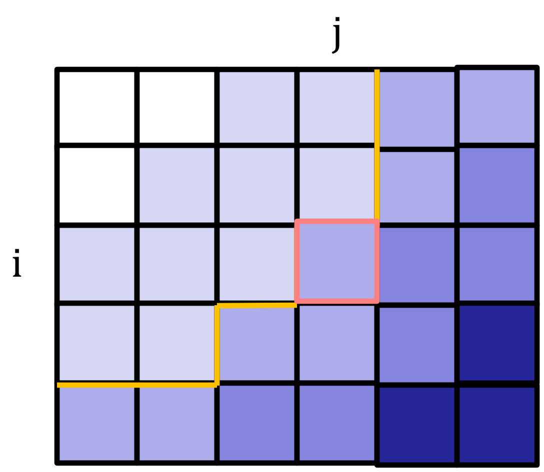
```
let i = 0;
let j = n;
... move j to left...
if (M[i][j] === x) return true;
```

{{ **Inv**: (x is not in row k for any $0 \le k \le i$) and

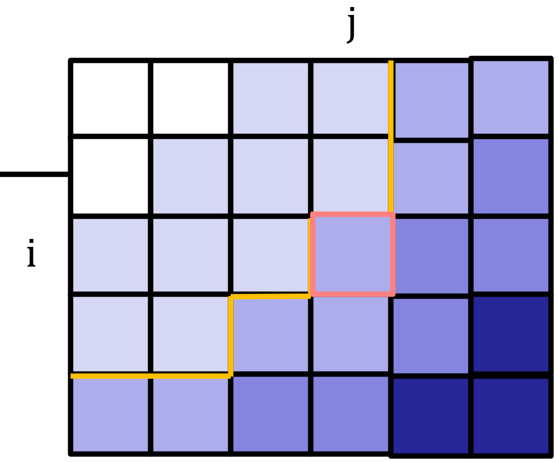
(M[i, k] < x for any $0 \le k < j$) and ($x \le$ M[i, k] for any $j \le k < n$) }}

```
while (i+1 !== n) {
   ...
}
return false;
```

Inv says we ruled out rows 0 .. i
and col j is line between _ < x and x ≤ _



Sorted Matrix Search




```
let i = 0;
let j = n;
... move j to left...
if (M[i][j] === x) return true;
```

{{ **Inv**: (x is not in row k for any $0 \le k \le i$) and

(M[i, k] < x for any $0 \le k < j$) and (x ≤ M[i, k] for any $j \le k < n$) }}

```
while (i+1 !== n) {
  i = i + 1;
  ... move j to the left...
  if (M[i][j] === x) return true;
}
return false;
```

We can avoid writing this code twice (without writing a separate function)...

Don't try this at home!

Sorted Matrix Search

```
let i = 0;
let j = n;
while (i !== n) {
  ... move j to left...
  if (M[i][j] === x) return true;
  {{ Inv: (x is not in row k for any 0 ≤ k ≤ i) and
        (M[i, k] < x for any 0 ≤ k < j) and (x ≤ M[i, k] for any j ≤ k < n) }}
  i = i + 1;
}
return false;
```

Loop condition was also changed

Inv is now checked in the middle of the loop!

Sorted Matrix Search

Final version is 9 lines of code.

Requires 6 lines of invariant assertions!

```
let i = 0;
let j = n;
while (i !== n) {
  {{ Inv: x ≤ M[i, k] for any j ≤ k < n }}
  while (j > 0 && x <= M[i][j-1])
    j = j - 1;
  if (M[i][j] === x)
    return true;
  {{ Inv: (x is not in row k for any 0 ≤ k ≤ i) and
        (M[i, k] < x for any 0 ≤ k < j) and (x ≤ M[i, k] for any j ≤ k < n) }}
  i = i + 1;
}
return false;
```