CSE 331

Reasoning About Array Indexing

James Wilcox

with thanks to Kevin Zatloukal for many slides

Indexing

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

$\text{at}(x :: L, 0) \quad := \; x$

$\text{at}(x :: L, n+1) \quad := \; \text{at}(L, n)$

- Retrieve an element of the list by <u>index</u>

- For example:

$\text{at}(1 :: 2 :: 3 :: 4 :: \text{nil}, 2)$

$= \text{at}(2 :: 3 :: 4 :: \text{nil}, 1)$ **def of** at

$= \text{at}(3 :: 4 :: \text{nil}, 0)$ **def of** at

$= 3$ **def of** at

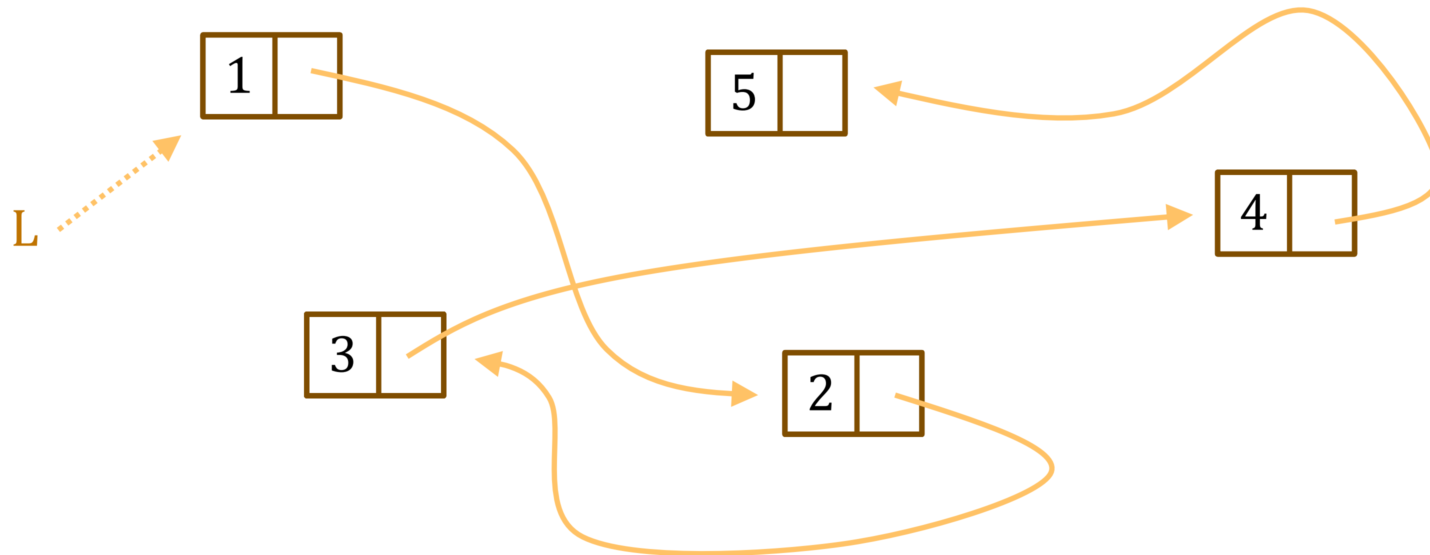
Indexing

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

$$\begin{aligned} \text{at}(x :: L\,,\, 0) &:= x \\ \text{at}(x :: L\,,\, n+1) &:= \text{at}(L, n) \end{aligned}$$

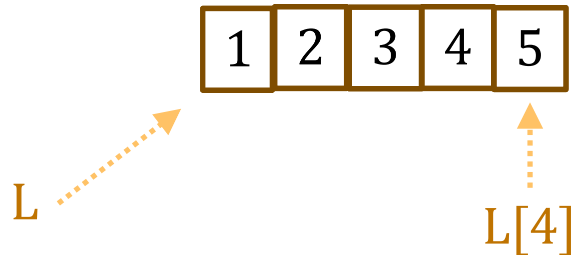
- **Use "$L[j]$" as an abbreviation for** $\text{at}(j, L)$
- **Not an efficient operation on lists…**

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

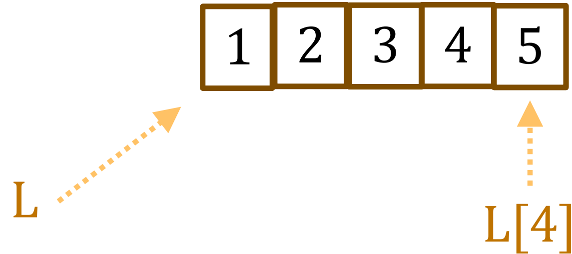


- Alternative: store the elements next to each other
  - can find the n-th entry by arithmetic:

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 L[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
  - **only checked at runtime in Java (no type checking)**

    TypeScript doesn't check it at all!!

Reasoning Toolkit

Description	Testing	Tools	Reasoning
no mutation	coverage	type checking	calculation induction
local variable mutation	“	“	Floyd logic
heap state mutation	“	“	rep invariants
array mutation	“	“	sublists for-any facts

Suffixes

$\text{suffix} : (\text{List}, \mathbb{N}) \to \text{List}$

$\text{suffix}(L, 0) := L$

$\text{suffix}(x :: L, n+1) := \text{suffix}(L, n)$

everything from here on not there yet

- **For example:**

$\text{suffix}(1 :: 2 :: 3 :: 4 :: \text{nil}, 2)$

$= \text{suffix}(2 :: 3 :: 4 :: \text{nil}, 1)$ — **def of** suffix

$= \text{suffix}(3 :: 4 :: \text{nil}, 0)$ — **def of** suffix

$= 3 :: 4 :: \text{nil}$ — **def of** suffix

  - **will use "**$L[j:]$**" as an abbreviation for** $\text{suffix}(L, j)$
  - $\text{suffix}(L, j)$ **is undefined if** $\text{len}(L) < j$

Prefixes

$\text{prefix} : (\text{List}, \mathbb{N}) \to \text{List}$

prefix(L , 0)	:=	nil	nothing more
prefix(x :: L , n+1)	:=	x :: prefix(L, n)	x is in the prefix

- **For example:**

prefix(1 :: 2 :: 3 :: 4 :: nil, 2)
 = 1 :: prefix(2 :: 3 :: 4 :: nil, 1) — **def of** prefix
 = 1 :: 2 :: prefix(3 :: 4 :: nil, 0) — **def of** prefix
 = 1 :: 2 :: nil — **def of** prefix

  - **will use "L[:j]" as an abbreviation for prefix(L, j)**
  - **note that L[:j] excludes j**

Prefixes & Suffixes

suffix(L , 0)	:= L	prefix(L , 0)	:= nil
suffix(x :: L , n+1)	:= suffix(L, n)	prefix(x :: L , n+1)	:= x :: prefix(L, n)

- **Can show that** prefix(L, j) ⧺ suffix(L, j) = L

Base Case: prefix(L, 0) ⧺ suffix(L, 0)

= nil ⧺ suffix(L, 0)	**def of** prefix
= nil ⧺ L	**def of** suffix
= L	

Inductive Step: prefix(x :: L, n+1) ⧺ suffix(x :: L, n+1)

= (x :: prefix(L, n)) ⧺ suffix(x :: L, n+1)	**def of** prefix
= (x :: prefix(L, n)) ⧺ suffix(L, n)	**def of** suffix
= x :: (prefix(L, n) ⧺ suffix(L, n))	**def of** concat
= x :: L	Ind. Hyp.

Prefixes & Suffixes

$$\begin{aligned} &\text{suffix}(L, 0) &&:= L && \text{prefix}(L, 0) &&:= \text{nil} \\ &\text{suffix}(x :: L, n+1) &&:= \text{suffix}(L, n) && \text{prefix}(x :: L, n+1) &&:= x :: \text{prefix}(L, n) \end{aligned}$$

- **Can show that** $\text{prefix}(L, j) \mathbin{+\!\!+} \text{suffix}(L, j) = L$
- **In shorthand, we have** $L[:j] \mathbin{+\!\!+} L[j:] = L$
  - **notation is inclusive on the left, exclusive on the right (Python style)**
  - **hence,** $L[:1] = [L[0]]$ **but** $L[:0] = \text{nil}$

Prefixes & Suffixes

$\text{suffix}(L, 0)$	$:= L$	$\text{prefix}(L, 0)$	$:= \text{nil}$
$\text{suffix}(x :: L, n{+}1)$	$:= \text{suffix}(L, n)$	$\text{prefix}(x :: L, n{+}1)$	$:= x :: \text{prefix}(L, n)$

- **In shorthand, we have** $L[:j] \mathbin{+\!\!\!+} L[j:] = L$
  - **notation is inclusive on the left, exclusive on the right (Python style)**

- **More useful facts about sublists:**

	Suffix	Prefix
Empty	$L[\text{len}(L):] = \text{nil}$	$L[:0] = \text{nil}$
Add One	$[L[j]] \mathbin{+\!\!\!+} L[j{+}1:] = L[j:]$	$L[:j] \mathbin{+\!\!\!+} [L[j]] = L[:j{+}1]$

  - **we will use these without explanation going forward…**

Loop Invariants With Indexing

Recall: Loop Correctness Example 2 (Topic 4)

$$\text{sum}(\text{nil}) := 0$$
$$\text{sum}(x :: L) := x + \text{sum}(L)$$

- **This loop claims to calculate it as well:**

```
public int sum(List L) {
  int s = 0;
  {{ Inv: sum(L_0) = s + sum(L) }}
  while (L != null) {
    s = s + L.hd;
    L = L.tl;
  }
  {{ s = sum(L_0) }}
  return s;
}
```

Change to a version that uses indexes…

Example 1: Sum of an Array

- Change to a version that uses an array and indexing

```
public int sum(int[] A) {
  int i = 0;
  int s = 0;
  {{ Inv: … }}
  while (i < A.length) {
    s = s + A[i];
    i = i + 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

Note that **A** does not change, just **i** and **s**

Example 1: Sum of an Array

- Change to a version that uses an array and indexing

```
public int sum(int[] A) {
  int i = 0;
  int s = 0;
  {{ Inv: s = sum(A[:i]) and i ≤ len(A) }}
  while (i < A.length) {
    s = s + A[i];
    i = i + 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

Important to know if A[i] is in the sum or not

Let's prove this correct!

Example 1: Sum of an Array

- Change to a version that uses an array and indexing

```
public int sum(int[] A) {
  int i = 0;
  int s = 0;
  {{ ____________________ }}
  {{ Inv: s = sum(A[:i]) and i ≤ len(A) }}
  while (i < A.length) {
    s = s + A[i];
    i = i + 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

Example 1: Sum of an Array

- Change to a version that uses an array and indexing

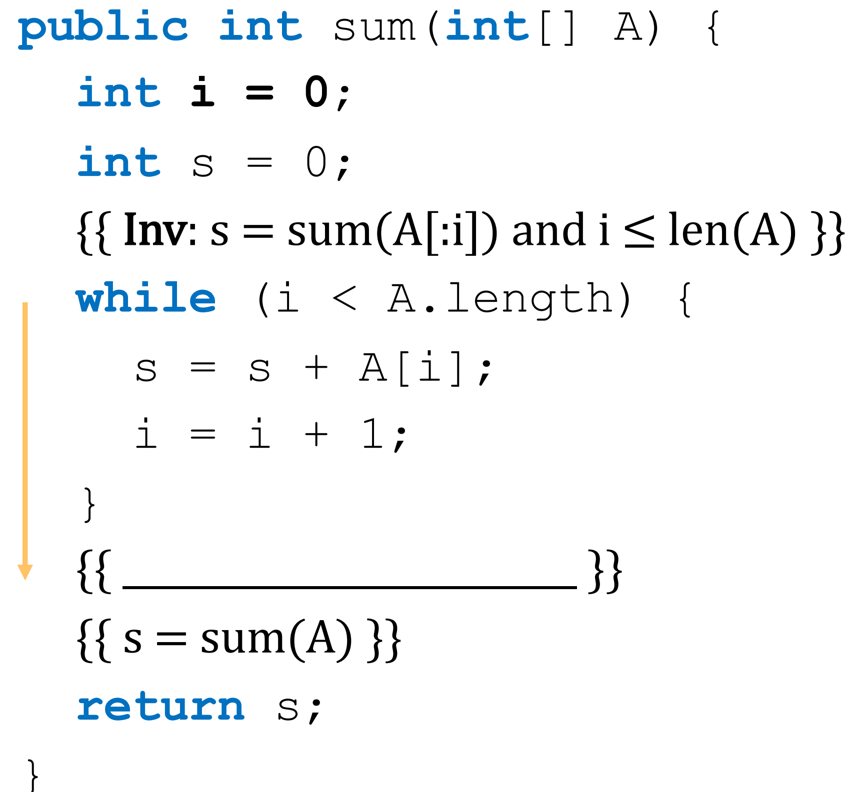
```
public int sum(int[] A) {
  int i = 0;
  int s = 0;
  {{ i = 0 and s = 0 }}
  {{ Inv: s = sum(A[:i]) and i ≤ len(A) }}
  while (i < A.length) {
    s = s + A[i];
    i = i + 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

$i = 0 \leq \text{len}(A)$

$\text{sum}(A[:i])$
$= \text{sum}(A[:0])$ **since** $i = 0$
$= \text{sum}(\text{nil})$ $A[:0] = \text{nil}$
$= 0$
$= s$

Example 1: Sum of an Array

- **Change to a version that uses an array and indexing**



Example 1: Sum of an Array

- Change to a version that uses an array and indexing

```
public int sum(int[] A) {
  int i = 0;
  int s = 0;
  {{ Inv: s = sum(A[:i]) and i ≤ len(A) }}
  while (i < A.length) {
    s = s + A[i];
    i = i + 1;
  }
  {{ s = sum(A[:i]) and i = len(A)}}
  {{ s = sum(A) }}
  return s;
}
```

sum(A[:i])
= sum(A[:len(A)]) since i = len(A)
= sum(A)

Example 1: Sum of an Array

- **Change to a version that uses an array and indexing**

```
public int sum(int[] A) {
  int i = 0;
  int s = 0;
  {{ Inv: s = sum(A[:i]) and i ≤ len(A) }}
  while (i < A.length) {
    {{ s = sum(A[:i]) and i < len(A) }}
    s = s + A[i];
    i = i + 1;
    {{ s = sum(A[:i]) and i ≤ len(A) }}
  }
  {{ s = sum(A) }}
  return s;
}
```

Example 1: Sum of an Array

- Change to a version that uses an array and indexing

```
public int sum(int[] A) {
  int i = 0;
  int s = 0;
  {{ Inv: s = sum(A[:i]) and i ≤ len(A) }}
  while (i < A.length) {
    {{ s = sum(A[:i]) and i < len(A) }}
    s = s + A[i];
    {{ ____________________ }}
    i = i + 1;
    {{ s = sum(A[:i]) and i ≤ len(A) }}
  }
  {{ s = sum(A) }}
  return s;
}
```

Example 1: Sum of an Array

- **Change to a version that uses an array and indexing**

```
public int sum(int[] A) {
  int i = 0;
  int s = 0;
  {{ Inv: s = sum(A[:i]) and i ≤ len(A) }}
  while (i < A.length) {
    {{ s = sum(A[:i]) and i < len(A) }}
    s = s + A[i];
    {{ s = sum(A[:i+1]) and i + 1 ≤ len(A) }}
    i = i + 1;
    {{ s = sum(A[:i]) and i ≤ len(A) }}
  }
  {{ s = sum(A) }}
  return s;
}
```

Example 1: Sum of an Array

- **Change to a version that uses an array and indexing**

```
public int sum(int[] A) {
  int i = 0;
  int s = 0;
  {{ Inv: s = sum(A[:i]) and i ≤ len(A) }}
  while (i < A.length) {
    {{ s = sum(A[:i]) and i < len(A) }}
    {{ ____________________ }}
    s = s + A[i];
    {{ s = sum(A[:i+1]) and i + 1 ≤ len(A) }}
    i = i + 1;
    {{ s = sum(A[:i]) and i ≤ len(A) }}
  }
  {{ s = sum(A) }}
  return s;
}
```

Example 1: Sum of an Array

- Change to a version that uses an array and indexing

```
public int sum(int[] A) {
  int i = 0;
  int s = 0;
  {{ Inv: s = sum(A[:i]) and i ≤ len(A) }}
  while (i < A.length) {
    {{ s = sum(A[:i]) and i < len(A) }}
    {{ s + A[i] = sum(A[:i+1]) and i + 1 ≤ len(A) }}
    s = s + A[i];
    {{ s = sum(A[:i+1]) and i + 1 ≤ len(A) }}
    i = i + 1;
    {{ s = sum(A[:i]) and i ≤ len(A) }}
  }
  {{ s = sum(A) }}
  return s;
}
```

Example 1: Sum of an Array

{{ $s = \text{sum}(A[:i])$ and $i < \text{len}(A)$ }}

{{ $s + A[i] = \text{sum}(A[:i+1])$ and $i + 1 \leq \text{len}(A)$ }}

$i \leq \text{len}(A) - 1$ **or equivalently**

$i + 1 \leq \text{len}(A)$

$$
\begin{aligned}
s + A[i] &= \text{sum}(A[:i]) + A[i] && \textbf{since } s = \text{sum}(A[:i]) \\
&= \text{sum}(A[:i]) + \text{sum}([A[i]]) && \textbf{def of } \text{sum}
\end{aligned}
$$

Recall: Length of Concatenated Lists

$$\begin{aligned} \text{concat}(\text{nil}, \text{R}) &:= \text{R} \\ \text{concat}(\text{x} :: \text{L}, \text{R}) &:= \text{x} :: \text{concat}(\text{L}, \text{R}) \end{aligned}$$

- **Proved that** $\text{len}(\text{concat}(\text{S}, \text{R})) = \text{len}(\text{S}) + \text{len}(\text{R})$
  - **prove by induction on** S
  - **prove the claim for any choice of** R (**i.e.,** R **is a variable**)

- **I.e., that** $\text{len}(\text{S} \mathbin{+\!\!+} \text{R}) = \text{len}(\text{S}) + \text{len}(\text{R})$

- **Can easily prove that** $\text{sum}(\text{S} \mathbin{+\!\!+} \text{R}) = \text{sum}(\text{S}) + \text{sum}(\text{R})$
  - **follows by induction on** S

Example 1: Sum of an Array

{{ s = sum(A[:i]) and i < len(A) }}

{{ s + A[i] = sum(A[:i+1]) and i + 1 ≤ len(A) }}

i ≤ len(A) – 1 **or equivalently**

i + 1 ≤ len(A)

s + A[i] = sum(A[:i]) + A[i] **since** s = sum(A[:i])

= sum(A[:i]) + sum([A[i]]) **def of** sum

= sum(A[:i] ⧺ [A[i]]) **by Lemma 1**

= sum(A[:i+1])

Example 1: Sum of an Array

- Change to a version that uses an array and indexing

```
public int sum(int[] A) {
  int i = 0;
  int s = 0;
  {{ Inv: s = sum(A[:i]) and i ≤ len(A) }}
  while (i < A.length) {
    {{ s = sum(A[:i]) and i < len(A) }}
    {{ s + A[i] = sum(A[:i+1]) and i + 1 ≤ len(A) }}
    s = s + A[i];
    i = i + 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

Done!

Example 2: Max of an Array

- **Can define the maximum of a list as follows:**

$$\mathrm{max} : (\mathrm{List}) \to \mathbb{Z}$$

$$
\begin{aligned}
&\mathrm{max}(x :: \mathrm{nil}) &&:= x && \\
&\mathrm{max}(x :: y :: L) &&:= x && \textbf{if } x > \mathrm{max}(y :: L) \\
&\mathrm{max}(x :: y :: L) &&:= \mathrm{max}(y :: L) && \textbf{if } x \leq \mathrm{max}(y :: L)
\end{aligned}
$$

  - **compares** x **to the maximum of the rest** $(y :: L)$

- **Note that this is undefined on** nil

Example 2: Max of an Array

- Can implement this with the following loop

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 1;
  int m = A[i];
  {{ Inv: ... }}
  while (i > 0) {
    if (A[i-1] >= m)
      m = A[i-1];
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

Should this by i >= 0?

Should this by A[i]?

Clear that up with the invariant

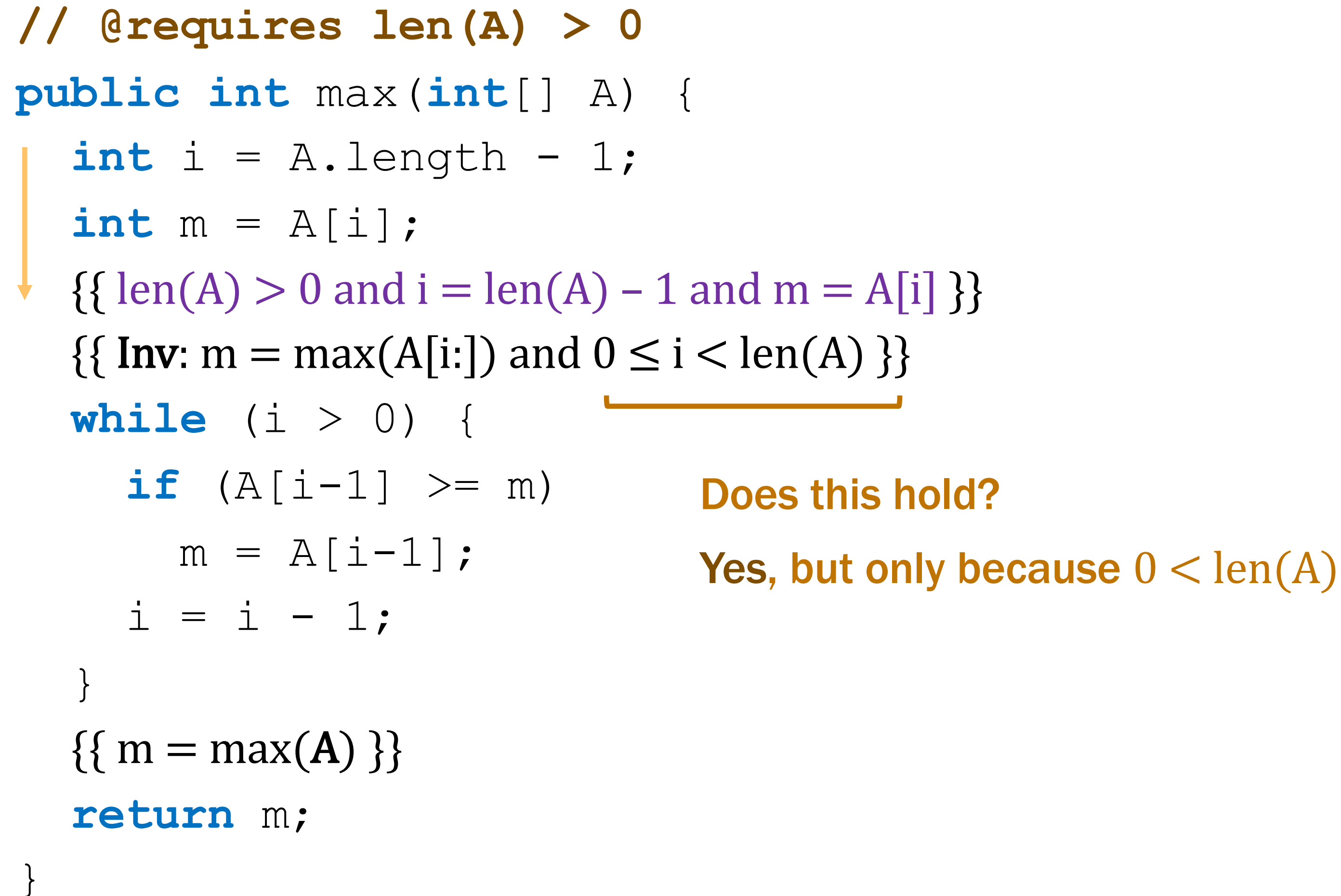
Example 2: Max of an Array

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 1;
  int m = A[i];
  {{ Inv: m = max(A[i:]) and 0 ≤ i < len(A) }}
  while (i > 0) {
    if (A[i-1] >= m)
      m = A[i-1];
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

Example 2: Max of an Array

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 1;
  int m = A[i];
  {{ ________________________ }}
  {{ Inv: m = max(A[i:]) and 0 ≤ i < len(A) }}
  while (i > 0) {
    if (A[i-1] >= m)
      m = A[i-1];
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

Example 2: Max of an Array

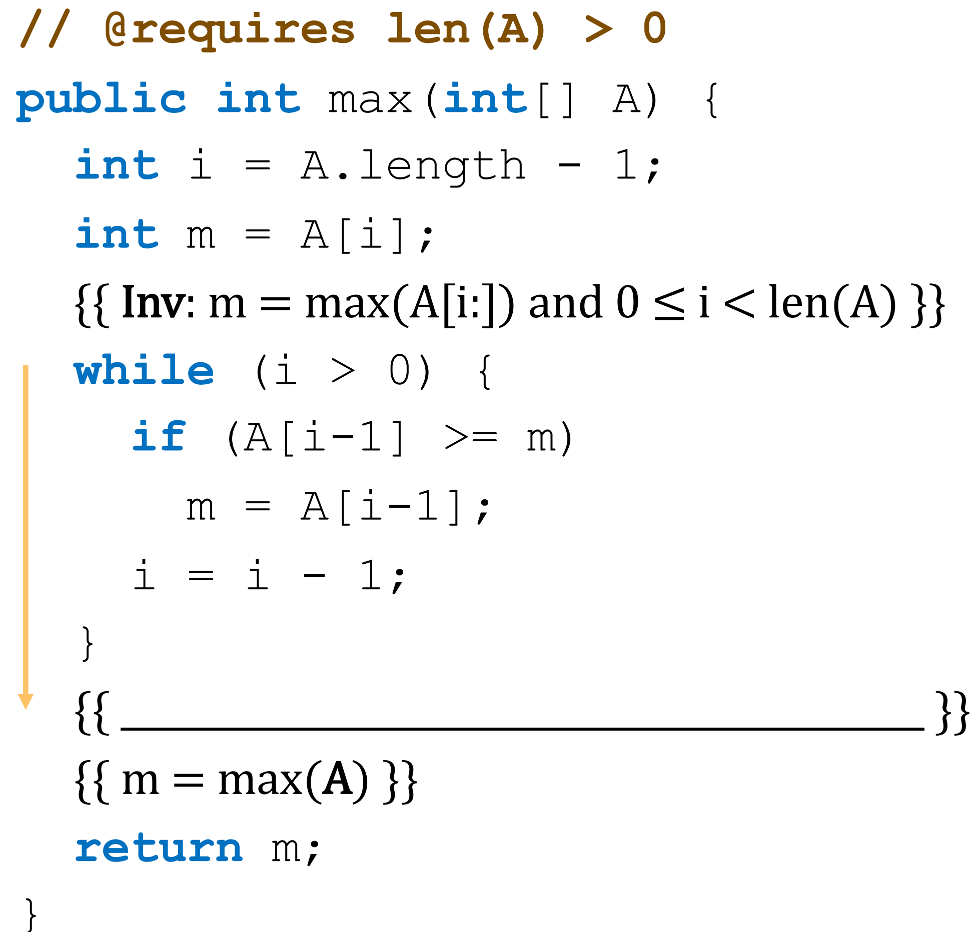


Example 2: Max of an Array

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 1;
  int m = A[i];
  {{ len(A) > 0 and i = len(A) – 1 and m = A[i] }}
  {{ Inv: m = max(A[i:]) and 0 ≤ i < len(A) }}
  while (i > 0) {
    if (A[i-1] >= m)
      m = A[i-1];
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

max(A[i:])
= max(A[len(A)-1:]) **since** i = len(A) – 1
= max([A[len(A) – 1]])
= A[len(A) – 1] **def of** max
= m

Example 2: Max of an Array



Example 2: Max of an Array

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 1;
  int m = A[i];
  {{ Inv: m = max(A[i:]) and 0 ≤ i < len(A) }}
  while (i > 0) {
    if (A[i-1] >= m)
      m = A[i-1];
    i = i - 1;
  }
  {{ m = max(A[i:]) and i = 0 }}
  {{ m = max(A) }}
  return m;
}
```

m = max(A[i:])
= max(A[0:]) since i = 0
= max(A)

Example 2: Max of an Array

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 1;
  int m = A[i];
  {{ Inv: m = max(A[i:]) and 0 ≤ i < len(A) }}
  while (i > 0) {
    {{ m = max(A[i:]) and 0 < i < len(A) }}
    if (A[i-1] >= m)
      m = A[i-1];
    i = i - 1;
    {{ m = max(A[i:]) and 0 ≤ i < len(A) }}
  }
  {{ m = max(A) }}
  return m;
}
```

Example 2: Max of an Array

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 1;
  int m = A[i];
  {{ Inv: m = max(A[i:]) and 0 ≤ i < len(A) }}
  while (i > 0) {
    {{ m = max(A[i:]) and 0 < i < len(A) }}
    if (A[i-1] >= m)
      m = A[i-1];
    {{ ______________________________ }}
    i = i - 1;
    {{ m = max(A[i:]) and 0 ≤ i < len(A) }}
  }
  {{ m = max(A) }}
  return m;
}
```

Example 2: Max of an Array

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 1;
  int m = A[i];
  {{ Inv: m = max(A[i:]) and 0 ≤ i < len(A) }}
  while (i > 0) {
    {{ m = max(A[i:]) and 0 < i < len(A) }}
    if (A[i-1] >= m)
      m = A[i-1];
    {{ m = max(A[i-1:]) and 0 ≤ i-1 < len(A) }}
    i = i - 1;
    {{ m = max(A[i:]) and 0 ≤ i < len(A) }}
  }
  {{ m = max(A) }}
  return m;
}
```

Example 2: Max of an Array

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 1;
  int m = A[i];
  {{ Inv: m = max(A[i:]) and 0 ≤ i < len(A) }}
  while (i > 0) {
    {{ m = max(A[i:]) and 0 < i < len(A) }}
    {{ ...                and 0 ≤ i-1 < len(A) }}
    if (A[i-1] >= m)
      m = A[i-1];
    {{ m = max(A[i-1:]) and 0 ≤ i-1 < len(A) }}
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

no changes to i

Does this hold?

Yes, but check both parts!

Example 2: Max of an Array

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 1;
  int m = A[i];
  {{ Inv: m = max(A[i:]) and 0 ≤ i < len(A) }}
  while (i > 0) {
    {{ m = max(A[i:]) and 0 < i < len(A) }}
    if (A[i-1] >= m)
      m = A[i-1];
    {{ m = max(A[i-1:]) and 0 ≤ i-1 < len(A) }}
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

Need to check other part
(Won't need facts about "i" anymore.)

Let's reason over paths...

Example 2: Max of an Array

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 1;
  int m = A[i];
  {{ Inv: m = max(A[i:]) and 0 ≤ i < len(A) }}
  while (i > 0) {
    {{ m = max(A[i:]) }}
    if (A[i-1] >= m)
      m = A[i-1];
    {{ ________________________ }}
    {{ m = max(A[i-1:]) }}
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

Case "else" (skip if)

Example 2: Max of an Array

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 1;
  int m = A[i];
  {{ Inv: m = max(A[i:]) and 0 ≤ i < len(A) }}
  while (i > 0) {
    {{ m = max(A[i:]) }}
    if (A[i-1] >= m)
      m = A[i-1];
    {{ m = max(A[i:]) and A[i-1] < m }}
    {{ m = max(A[i-1:]) }}
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

Case "else" (skip if)

$A[i-1] < m = max(A[i:])$

$max(A[i-1:])$
$= max(A[i-1] :: A[i:])$
$= max(A[i:])$ **def of** max **(since** $A[i] < max(A[i:])$**)**
$= m$

Example 2: Max of an Array

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 1;
  int m = A[i];
  {{ Inv: m = max(A[i:]) and 0 ≤ i < len(A) }}
  while (i > 0) {
    {{ m = max(A[i:]) }}
    if (A[i-1] >= m)
      m = A[i-1];
    {{ ________________________________ }}
    {{ m = max(A[i-1:]) }}
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

Case "then" (enter "if")

Example 2: Max of an Array

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 1;
  int m = A[i];
  {{ Inv: m = max(A[i:]) and 0 ≤ i < len(A) }}
  while (i > 0) {
    {{ m = max(A[i:]) }}
    if (A[i-1] >= m)
      m = A[i-1];
    {{ m₀ = max(A[i:]) and A[i-1] ≥ m₀ and m = A[i-1] }}
    {{ m = max(A[i-1:]) }}
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

$A[i-1] \geq m_0 = \max(A[i:])$

$\max(A[i-1:])$
$= \max(A[i-1] :: A[i:])$
$= A[i-1]$ — **def of** max **(since** $A[i] \geq \max(A[i:])$**)**
$= m$

Takeaways

- Reasoning about arrays requires new tools
  - prefixes and suffixes now (more later)
- Trying move faster…
- But the tools still work!
  - can return to them when it gets tricky
  - always fine to slow down and check carefully
- The tools help you know what needs checking

From Invariant To Code

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 Example 1: Sum of an Array

```
public int sum(int[] A) {
  int i = 0;
  int s = 0;
  {{ Inv: s = sum(A[:i]) and i ≤ len(A) }}
  while (i < A.length) {
    s = s + A[i];
    i = i + 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

Example 4: Sum of an Array (Version 2)

```
public int sum(int[] A) {
  int i = ______
  int s = 0;
  {{ Inv: s = sum(A[:i+1]) }}
  while (________________________) {

    ___________
    i = i + 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

Idea: add from front to back, but now i is the last index included in the sum

Example 4: Sum of an Array (Version 2)

```
public int sum(int[] A) {
  int i = ______
  int s = 0;
  {{ Inv: s = sum(A[:i+1]) }}
  while (________________________) {
    ___________
    i = i + 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

How do we initialize i so that sum(A[:i+1]) = 0**?**

Choose i **so that** A[:i+1] = nil **since** sum(nil) = 0

We need i = -1 **since** A[:0] = nil

Example 4: Sum of an Array (Version 2)

```
public int sum(int[] A) {
  int i = -1;
  int s = 0;
  {{ Inv: s = sum(A[:i+1]) }}
  while (________________________) {
    __________
    i = i + 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

Example 4: Sum of an Array (Version 2)

```
public int sum(int[] A) {
  int i = -1;
  int s = 0;
  {{ Inv: s = sum(A[:i+1]) }}
  while (________________________) {
    ___________
    i = i + 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

How do we exit so that $\text{sum}(A[:i+1]) = \text{sum}(A)$**?**

Choose exit so that $A[:i+1] = A$

We need $i = \text{len}(A) - 1$

Loop while $i < \text{len}(A) - 1$

Example 4: Sum of an Array (Version 2)

```
public int sum(int[] A) {
  int i = -1;
  int s = 0;
  {{ Inv: s = sum(A[:i+1]) and -1 ≤ i ≤ len(A) - 1 }}
  while (i < A.length - 1) {
    __________
    i = i + 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

Can now fill in the range of values for i.
(Important to avoid array out-of-bounds.)

Example 4: Sum of an Array (Version 2)

```
public int sum(int[] A) {
  int i = -1;
  int s = 0;
  {{ Inv: s = sum(A[:i+1]) and -1 ≤ i ≤ len(A) - 1 }}
  while (i < A.length - 1) {

    ____________
    i = i + 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

How do we update s?

Useful to do one step of Floyd logic...

Example 4: Sum of an Array (Version 2)

```
public int sum(int[] A) {
  int i = -1;
  int s = 0;
  {{ Inv: s = sum(A[:i+1]) and -1 ≤ i ≤ len(A) - 1 }}
  while (i < A.length - 1) {
    {{ s = sum(A[:i+1]) }}
    ____________
    {{ s = sum(A[:i+2]) }}
    i = i + 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

How do we change s to go from sum(A[:i+1]) to sum(A[:i+2])?

Need to add in A[i+1] (Not A[i]!)

Example 4: Sum of an Array (Version 2)

```
public int sum(int[] A) {
  int i = -1;
  int s = 0;
  {{ Inv: s = sum(A[:i+1]) and -1 ≤ i ≤ len(A) - 1 }}
  while (i < A.length - 1) {
    {{ s = sum(A[:i+1]) }}
    s = s + A[i+1];
    {{ s = sum(A[:i+2]) }}
    i = i + 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

How do know this is in bounds?

Let's fill in our information about i...

Example 4: Sum of an Array (Version 2)

```
public int sum(int[] A) {
  int i = -1;
  int s = 0;
  {{ Inv: s = sum(A[:i+1]) and -1 ≤ i ≤ len(A) - 1 }}
  while (i < A.length - 1) {
    {{ s = sum(A[:i+1]) and -1 ≤ i < len(A) - 1 }}
    s = s + A[i+1];
    i = i + 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

Why is that what we know?

Get $-1 \le i$ from Inv and $i < \text{len}(A) - 1$ from the loop condition.

How do know $A[i+1]$ is in bounds?

$-1 \le i < \text{len}(A) - 1$ implies $0 \le i+1 < \text{len}(A)$

Example 4: Sum of an Array (Version 2)

```
public int sum(int[] A) {
  int i = -1;
  int s = 0;
  {{ Inv: s = sum(A[:i+1]) and -1 ≤ i ≤ len(A) - 1 }}
  while (i < A.length - 1) {
    s = s + A[i+1];
    i = i + 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

Done!

Example 5: Sum of an Array (Version 3)

```
public int sum(int[] A) {
  int i = ______
  int s = 0;
  {{ Inv: s = sum(A[i:]) }}
  while (________________________) {
    ___________
    i = i - 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

Idea: add from back to front, with i included in the sum

Example 5: Sum of an Array (Version 3)

```
public int sum(int[] A) {
  int i = ______
  int s = 0;
  {{ Inv: s = sum(A[i:]) }}
  while (________________________) {
    ___________
    i = i - 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

How do we initialize i so that $\text{sum}(A[i:]) = 0$?

Choose i so that $A[i:] = \text{nil}$ since $\text{sum}(\text{nil}) = 0$

We need $i = \text{len}(A)$ since $A[\text{len}(A):] = \text{nil}$

Example 5: Sum of an Array (Version 3)

```
public int sum(int[] A) {
  int i = A.length;
  int s = 0;
  {{ Inv: s = sum(A[i:]) }}
  while (________________________) {
    ___________
    i = i - 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

Example 5: Sum of an Array (Version 3)

```
public int sum(int[] A) {
  int i = A.length;
  int s = 0;
  {{ Inv: s = sum(A[i:]) }}
  while (______________________) {
    __________
    i = i - 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

How do we exit so that $\mathrm{sum}(A[i:]) = \mathrm{sum}(A)$**?**

Choose exit so that $A[i:] = A$

We need $i = 0$

Loop while $i > 0$

Example 5: Sum of an Array (Version 3)

```
public int sum(int[] A) {
  int i = A.length;
  int s = 0;
  {{ Inv: s = sum(A[i:]) }}
  while (i > 0) {
    __________
    i = i - 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

Example 5: Sum of an Array (Version 3)

```
public int sum(int[] A) {
  int i = A.length;
  int s = 0;
  {{ Inv: s = sum(A[i:]) and 0 ≤ i ≤ len(A) }}
  while (i > 0) {
    __________
    i = i - 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

This is the range of values for i.
(Note that 0 is included!)

Example 5: Sum of an Array (Version 3)

```
public int sum(int[] A) {
  int i = A.length;
  int s = 0;
  {{ Inv: s = sum(A[i:]) and 0 ≤ i ≤ len(A) }}
  while (i > 0) {
    __________
    i = i - 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

How do we update s?

Let's do one step Floyd logic…

Example 5: Sum of an Array (Version 3)

```
public int sum(int[] A) {
  int i = A.length;
  int s = 0;
  {{ Inv: s = sum(A[i:]) and 0 ≤ i ≤ len(A) }}
  while (i > 0) {
    {{ s = sum(A[i:]) }}
    ___________
    {{ ______________ }}
    i = i - 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

What goes here?

Example 5: Sum of an Array (Version 3)

```
public int sum(int[] A) {
  int i = A.length;
  int s = 0;
  {{ Inv: s = sum(A[i:]) and 0 ≤ i ≤ len(A) }}
  while (i > 0) {
    {{ s = sum(A[i:]) }}
    ____________
    {{ s = sum(A[i-1:]) }}
    i = i - 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

How do we change s to go from sum(A[i:]) to sum(A[i-1:])?

Need to add in A[i-1] (Not A[i]!)

Example 5: Sum of an Array (Version 3)

```
public int sum(int[] A) {
  int i = A.length;
  int s = 0;
  {{ Inv: s = sum(A[i:]) and 0 ≤ i ≤ len(A) }}
  while (i > 0) {
    {{ s = sum(A[i:]) }}
    s = s + A[i-1];
    {{ s = sum(A[i-1:]) }}
    i = i - 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

Is this array access in bounds?

Example 5: Sum of an Array (Version 3)

```
public int sum(int[] A) {
  int i = A.length;
  int s = 0;
  {{ Inv: s = sum(A[i:]) and 0 ≤ i ≤ len(A) }}
  while (i > 0) {
    {{ s = sum(A[i:]) and 0 < i ≤ len(A) }}
    s = s + A[i-1];
    i = i - 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

Why is this what we get from Floyd logic?

Why does this tell us the array access is within bounds?

Example 5: Sum of an Array (Version 3)

```
public int sum(int[] A) {
  int i = A.length;
  int s = 0;
  {{ Inv: s = sum(A[i:]) and 0 ≤ i ≤ len(A) }}
  while (i > 0) {
    s = s + A[i-1];
    i = i - 1;
  }
  {{ s = sum(A) }}
  return s;
}
```

Done!

Recall: Example 2: Max of an Array

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = …;
  int m = …;
  {{ Inv: m = max(A[i:]) }}
  while (i > 0) {
    if (A[i-1] >= m)
      m = A[i-1];
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

Example 6: Max of an Array (Version 2)

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = ____________
  int m = ____________
  {{ Inv: m = max(A[i+1:]) }}
  while (_____________) {
    __________
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

Idea: work from back to front, with i not included in value of m

Example 6: Max of an Array (Version 2)

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = ______________
  int m = ______________
  {{ Inv: m = max(A[i+1:]) }}
  while (_______________) {
    ____________
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

What is the easiest way to make this hold initially?

Remember that max is undefined on nil

Easiest is $m = \max([A[\text{len}(A)-1]])$

How do we initialize i so that $m = \max(A[i+1:])$

We need $A[i+1:] = [A[\text{len}(A)-1]]$

Must take $i = \text{len}(A)\text{-}2$

Example 6: Max of an Array (Version 2)

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 2;
  int m = A[i+1];
  {{ Inv: m = max(A[i+1:]) }}
  while (________________) {
    ____________
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

Example 6: Max of an Array (Version 2)

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 2;
  int m = A[i+1];
  {{ Inv: m = max(A[i+1:]) }}
  while (________________) {
    ____________
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

How do we exit so that $\max(A[i+1:]) = \max(A)$**?**

Choose exit so that $A[i+1:] = A$

We need $i = -1$

Loop while $i > -1$ **or in other words** $i \geq 0$

Example 6: Max of an Array (Version 2)

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 2;
  int m = A[i+1];
  {{ Inv: m = max(A[i+1:]) }}
  while (i >= 0) {
    ____________
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

Can now fill in the range of i

Example 6: Max of an Array (Version 2)

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 2;
  int m = A[i+1];
  {{ Inv: m = max(A[i+1:]) and -1 ≤ i ≤ len(A)-2 }}
  while (i >= 0) {
    ____________
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

Example 6: Max of an Array (Version 2)

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 2;
  int m = A[i+1];
  {{ Inv: m = max(A[i+1:]) and -1 ≤ i ≤ len(A)-2 }}
  while (i >= 0) {
    ____________
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

How do we update s?

Let's do one step Floyd logic...

Example 6: Max of an Array (Version 2)

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 2;
  int m = A[i+1];
  {{ Inv: m = max(A[i+1:]) and -1 ≤ i ≤ len(A)-2 }}
  while (i >= 0) {
    {{ m = max(A[i+1:]) }}
    ____________
    {{ ________________ }}
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

What goes here?

Example 6: Max of an Array (Version 2)

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 2;
  int m = A[i+1];
  {{ Inv: m = max(A[i+1:]) and -1 ≤ i ≤ len(A)-2 }}
  while (i >= 0) {
    {{ m = max(A[i+1:]) }}
    ____________
    {{ m = max(A[i:]) }}
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

How do we implement this?

How do we change s to go from $\max(A[i+1:])$ to $\max(A[i:])$?

Need to look at $A[i]$

If $A[i] > m$, then $A[i]$ is new max

Otherwise, m is still the max

$$
\begin{aligned}
\max(x :: \text{nil}) &:= x \\
\max(x :: y :: L) &:= x && \text{if } x > \max(y :: L) \\
\max(x :: y :: L) &:= \max(y :: L) && \text{if } x \leq \max(y :: L)
\end{aligned}
$$

Example 6: Max of an Array (Version 2)

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 2;
  int m = A[i+1];
  {{ Inv: m = max(A[i+1:]) and -1 ≤ i ≤ len(A)-2 }}
  while (i >= 0) {
    {{ m = max(A[i+1:]) }}
    if (A[i] > m)
      m = A[i];
    {{ m = max(A[i:]) }}
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

Example 6: Max of an Array (Version 2)

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 2;
  int m = A[i+1];
  {{ Inv: m = max(A[i+1:]) and -1 ≤ i ≤ len(A)-2 }}
  while (i >= 0) {
    {{ m = max(A[i+1:]) }}
    if (A[i] > m)
      m = A[i];
    {{ m = max(A[i:]) }}
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

Are we sure this is in bounds?

Let's reason forward to check...

Example 6: Max of an Array (Version 2)

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 2;
  int m = A[i+1];
  {{ Inv: m = max(A[i+1:]) and -1 ≤ i ≤ len(A)-2 }}
  while (i >= 0) {
    if (A[i] > m)
      {{ m = max(A[i+1:]) and 0 ≤ i ≤ len(A)-2 }}
      m = A[i];
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

Why is that what we get?

Why does that tell us this is okay?

Example 6: Max of an Array (Version 2)

```
// @requires len(A) > 0
public int max(int[] A) {
  int i = A.length - 2;
  int m = A[i+1];
  {{ Inv: m = max(A[i+1:]) and -1 ≤ i ≤ len(A)-2 }}
  while (i >= 0) {
    if (A[i] > m)
      m = A[i];
    i = i - 1;
  }
  {{ m = max(A) }}
  return m;
}
```

Done!

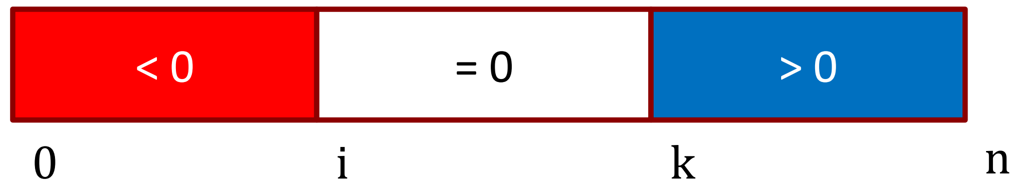
Example 7: Sorting Negative, Zero, Positive

- Reorder an array so that
  - negative numbers come first, then zeros, then positives
  - famous coding interview question (decades ago)

```
/**
 * 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
 */
public static int[] sortPosNeg(int[] A)
```

Example 7: 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 is the *idea* of our loop?**

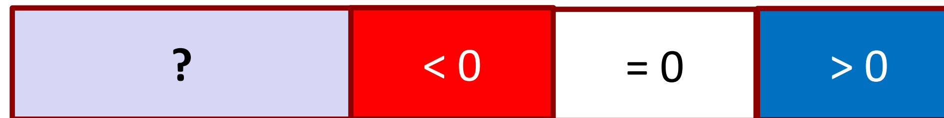
how do we represent <u>partial</u> progress in our invariant?

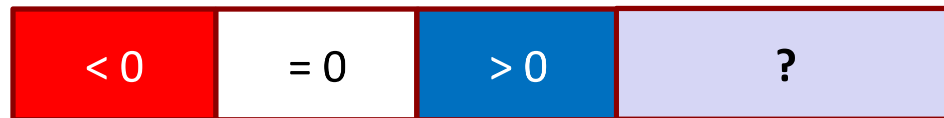
Example 7: Sorting Negative, Zero, Positive

How do we represent partial progress in our invariant?

- needs allow elements with *unknown* values

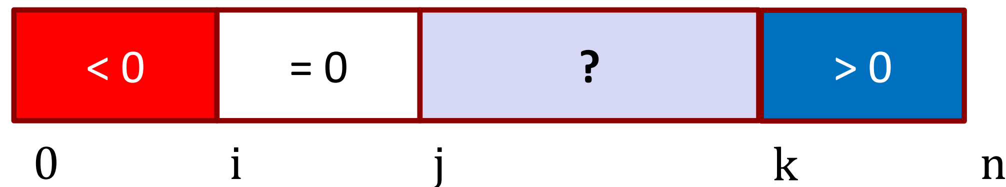
    initially, we don't know anything about the array values





Example 7: Sorting Negative, Zero, Positive

Our Invariant:



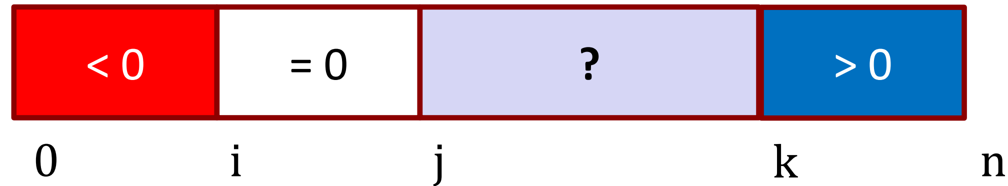
$A[\ell] < 0$ for any $0 \leq \ell < i$

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

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

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

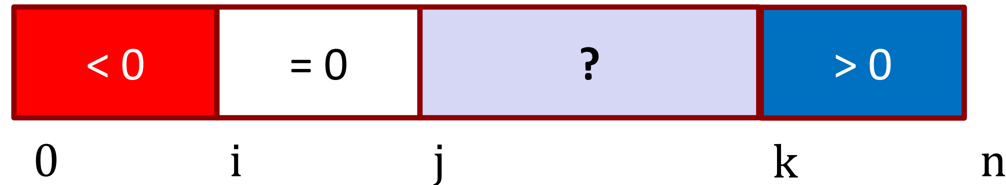
Example 7: Sorting Negative, Zero, Positive



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

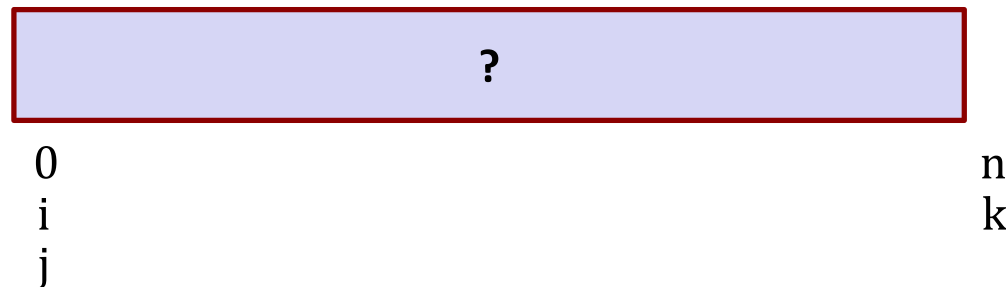
- Figure out the code for
  - how to initialize
  - when to exit
  - loop body

Example 7: Sorting Negative, Zero, Positive

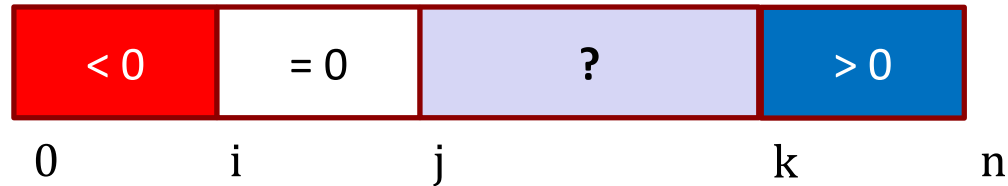


- **Will have variables i, j, and k with $i \leq j \leq k$**

- **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 $k = n$

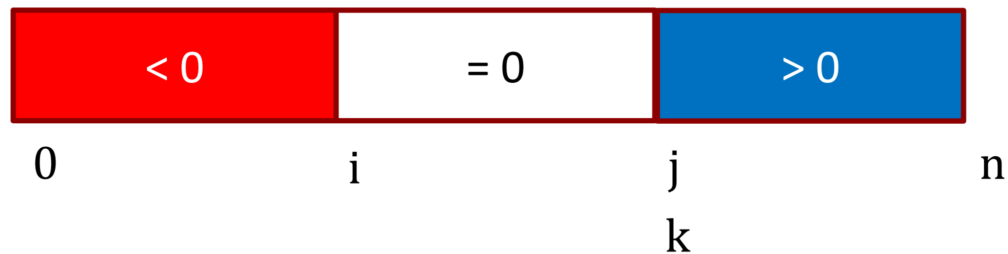


Example 7: Sorting Negative, Zero, Positive



- **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

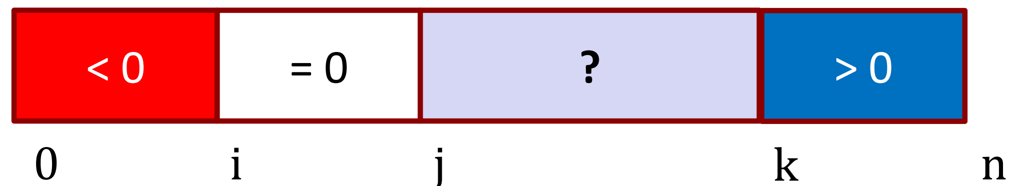
```
int i = 0;
int j = 0;
int k = A.length;
```

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

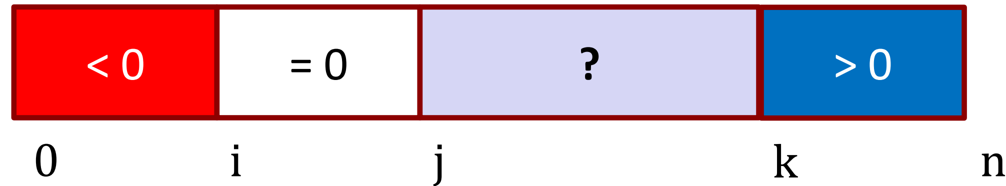
```
while (j < k) {
   ...
}
```

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

```
return new int[] {i, j};
```



Example 7: Sorting Negative, Zero, Positive

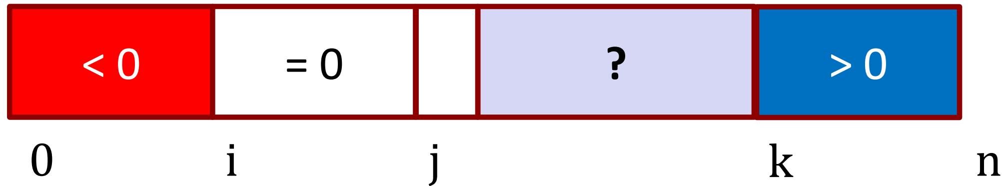


- **How do we make progress?**
  - **try to increase j by 1 or decrease k by 1**

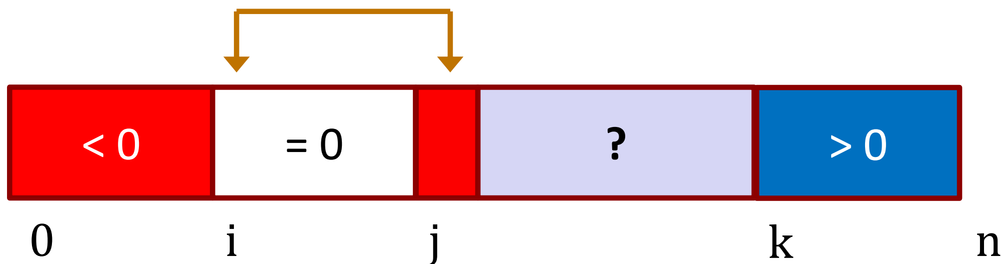
- **Look at $A[j]$ and figure out where it goes**

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

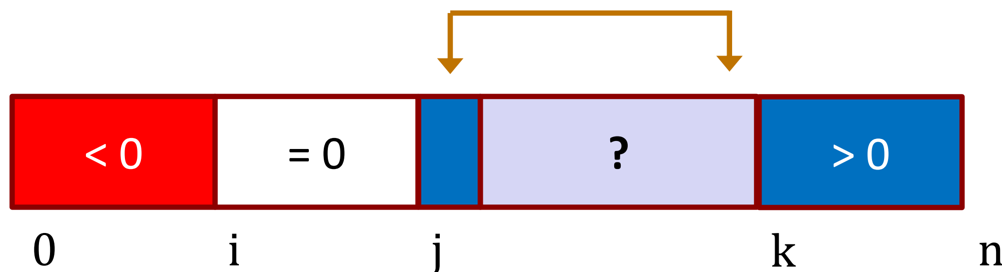
Example 7: Sorting Negative, Zero, Positive



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 $k = k_0 - 1$

Example 7: Sort Positive, Zero, Negative

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

```
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;
  }
}
```

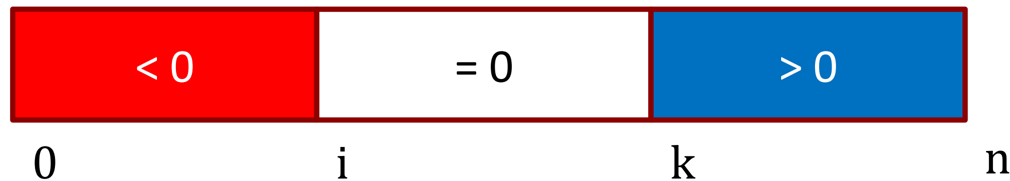
Recall: Reasoning Toolkit

Description	Testing	Tools	Reasoning
no mutation	coverage	type checking	calculation induction
local variable mutation	“	“	Floyd logic
heap state mutation	“	“	rep invariants
array mutation	“	“	sublists for-any facts

"For Any" Facts

Recall: Sorting Negative, Zero, Positive

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



– could write first region as $\text{negative}(A[:i])$ and
  last region as $\text{positive}(A[k:])$
– but middle region is not a prefix or suffix

"For Any" Facts

- Necessary facts are not always prefix / suffix
  - may have facts about arbitrary parts of the array
- For example, what does "A is sorted" mean?
  - we have $A[0] < A[1] < \ldots < A[n\text{-}1]$ or more formally:

$$A[\mathbf{j}] < A[\mathbf{j}+1] \quad \text{for any} \quad 0 \le \mathbf{j} \le \text{len}(A) - 2$$

- We will call these "for any" facts
  - they are more cumbersome than simple equations
  - they will show up inside individual assertions now!

    "with great power comes great responsibility"…

Example 8: Binary Search

- Binary search looks for a number in a sorted array
  - return true if $A[i] = y$ for some i

```
// @requires A[i] < A[i+1] for any 0 <= i < len(A)-1
// @returns false if A[i] != y for any 0 <= i < len(A)
//          true  otherwise
public boolean bsearch(int[] A, int y) { … }
```

Example 8: Binary Search

- Invariant of binary search has "for any" facts

```
public boolean bsearch(int[] A, int y) { // req A sorted
  int j = 0;
  int k = A.length;
  {{ Inv: (A[i] < y  for any  0 ≤ i < j) and (y < A[i]  for any  k ≤ i < len(A)) }}
  while (j < k) {
    int m = (j + k) / 2;
    if (y < A[m]) {
      k = m;
    } else if (A[m] < y) {
      j = m + 1;
    } else {
      return true;
    }
  }
  return false;
}
```

Why the asymmetry?

Example 8: Binary Search

- Invariant of binary search has "for any" facts

```
public boolean bsearch(int[] A, int y) { // req A sorted
  int j = 0;
  int k = A.length;
  {{ j = 0 and k = len(A) }}
  {{ Inv: (A[i] < y  for any  0 ≤ i < j) and (y < A[i]  for any  k ≤ i < len(A)) }}
  while (j < k) {
    int m = (j + k) / 2;
    if (y < A[m]) {
      k = m;
    } else if (A[m] < y) {
      j = m + 1;
    } else {
      return true;
    }
  }
  return false;
}
```

No integer i satisfies $0 \le i < 0$

No integer i satisfies $\text{len}(A) \le i < \text{len}(A)$

Example 8: Binary Search

- Invariant of binary search has "for any" facts

{{ **Inv**: (A[i] $<$ y for any $0 \le i < j$) and (y $<$ A[i] for any $k \le i < len(A)$) }}

```
while (j < k) {
  int m = (j + k) / 2;
  if (y < A[m]) {
    k = m;
  } else if (A[m] < y) {
    j = m + 1;
  } else {
    return true;
  }
}
```

{{ Inv and $j = k$ }}

{{ A[i] $\ne$ y for any $0 \le i < len(A)$ }}

```
return false;
}
```

Every index $i < j$ is smaller than y

Every index $k \le i$ is larger than y

Example 8: Binary Search

{{ **Inv**: $(A[i] < y$ for any $0 \le i < j)$ and $(y < A[i]$ for any $k \le i < \text{len}(A))$ }}

```
while (j < k) {
  int m = (j + k) / 2;
  if (y < A[m]) {
    k = m;
  } else if (A[m] < y) {
    j = m + 1;
  } else {
    return true;
  }
}
return false;
}
```

Work through cases…

Example 8: Binary Search

```
  {{ Inv: (A[i] < y  for any  0 ≤ i < j) and (y < A[i]  for any  k ≤ i < len(A)) }}
  while (j < k) {
    int m = (j + k) / 2;
    if (y < A[m]) {
      {{ Inv and y < A[m] }}
      k = m;
    } else if (A[m] < y) {
      {{ Inv and A[m] < y }}
      j = m + 1;
    } else {
      {{ Inv and A[m] = y }}
      return true;
    }
  }
  return false;
}
```

Example 8: Binary Search

```
{{ Inv: (A[i] < y  for any  0 ≤ i < j) and (y < A[i]  for any  k ≤ i < len(A)) }}
while (j < k) {
  int m = (j + k) / 2;
  if (y < A[m]) {
    {{ Inv and y < A[m] }}
    k = m;
  } else if (A[m] < y) {
    {{ Inv and A[m] < y }}
    j = m + 1;
  } else {
    {{ Inv and A[m] = y }}
    {{ A[i] = y for some i }}
    return true;
  }
}
return false;
}
```

True. Take $i = m$

Example 8: Binary Search

{{ **Inv**: ($A[i] < y$ for any $0 \le i < j$) and ($y < A[i]$ for any $k \le i < len(A)$) }}

```
while (j < k) {
  int m = (j + k) / 2;
  if (y < A[m]) {
    {{ Inv and y < A[m] }}
    {{ (A[i] < y  for any  0 ≤ i < j) and (y < A[i]  for any  m ≤ i < len(A)) }}
    k = m;
  } else if (A[m] < y) {
    {{ Inv and A[m] < y }}
    j = m + 1;
  } else {
    return true;
  }
}
return false;
}
```

Left part is known from Inv.

We know that $y < A[m]$ holds.

Why do we know $y < A[i]$ holds for $m < i$?

Holds because A is sorted.

Example 8: Binary Search

```
{{ Inv: (A[i] < y  for any  0 ≤ i < j) and (y < A[i]  for any  k ≤ i < len(A)) }}
while (j < k) {
  int m = (j + k) / 2;
  if (y < A[m]) {
    {{ Inv and y < A[m] }}
    k = m;
  } else if (A[m] < y) {
    {{ Inv and A[m] < y }}
    {{ (A[i] < y  for any  0 ≤ i < m+1) and (y < A[i]  for any  k ≤ i < len(A)) }}
    j = m + 1;
  } else {
    return true;
  }
}
return false;
}
```

Right part is known from Inv.

We know that $y < A[m]$ holds.

Have $y < A[i]$ holds for $i < m$ since A is sorted.

Mutation of Arrays

Mutation With For-Any Facts

- Mutation makes "for any" facts change
  - this can happen line by line!
  - working forward:

{{ $A[j] < A[j+1]$ for any $0 \le j \le 9$ }}

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

{{ ______________________ }}

Mutation With For-Any Facts

- Mutation makes "for any" facts change
  - this can happen line by line!
  - working forward:

{{ $A[j] < A[j+1]$ for any $0 \le j \le 9$ }}

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

{{ ($A[j] < A[j+1]$ for any $\mathbf{1} \le j \le 9$) and $A[0] = 100$ }}

  - old facts about A[0] are invalidated

    if we wanted to, we could note that $A_0[0] \le A[1]$

Mutation With For-Any Facts

- Mutation makes "for any" facts change
  - this can happen line by line!
  - working backward:

{{ ________________________________ }}

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

{{ $A[j] < A[j+1]$ for any $0 \le j \le 9$ }}

Mutation With For-Any Facts

- **Mutation makes "for any" facts change**
  - this can happen line by line!
  - working backward:

{{ $100 < A[1]$ and ($A[j] < A[j+1]$ for any $1 \leq j \leq 9$) }}

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

{{ $A[j] < A[j+1]$ for any $0 \leq j \leq 9$ }}

  - just substitution as usual

    but be careful about finding everywhere $A[0]$ appears

Mutation With For-Any Facts

- Mutation makes "for any" facts change
  - this can happen line by line!

- Easiest if we mutate the end value in the range
  - but other mutations are possible
  - working forward:

{{ $A[j] < A[j+1]$ for any $0 \leq j \leq 9$ }}

```
A[5] = 100;
```

{{ ______________________ }}

Mutation With For-Any Facts

- Mutation makes "for any" facts change
  - this can happen line by line!

- Easiest if we mutate the end value in the range
  - but other mutations are possible
  - working forward:

{{ $A[j] < A[j+1]$ for any $0 \le j \le 9$ }}

```
A[5] = 100;
```

{{ ($A[j] < A[j+1]$ for any $0 \le j \le 4$) and $A[5] = 100$ and
($A[j] < A[j+1]$ for any $6 \le j \le 9$) }}

For-any facts get out of hand quickly!

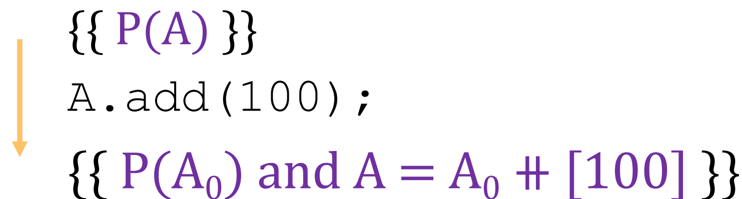
We will look at ways to cope in Topic 10

Adding & Removing Elements

- **Can efficiently add & remove at end of array:**

```
ArrayList<Integer> A = …;
A.add(100);
```

- **How do we handle this in Floyd logic?**
  - **working forward:**




```
// @modifies obj
// @effects obj = obj_0 ++ [n]
void add(int n)
```

  - **value changes by appending $[100]$ to previous value**

Adding & Removing Elements

- **Can efficiently add & remove at end of array:**

```
ArrayList<Integer> A = …;
A.add(100);
```

- **How do we handle this in Floyd logic?**
  - **working backward:**

$\{\{\ P(A +\!\!\!+ [100])\ \}\}$

```
A.add(100);
```

$\{\{\ P(A)\ \}\}$

```
// @modifies obj
// @effects obj = obj_0 ++ [n]
void add(int n)
```

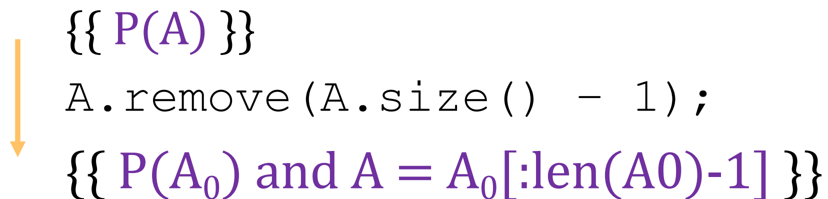
  - **just substitution through assignment** $A = A +\!\!\!+ [100]$

Adding & Removing Elements

- **Can efficiently add & remove at end of array:**

```
ArrayList<Integer> A = …;
A.remove(A.size() - 1);  // remove last element
```

- **How do we handle this in Floyd logic?**
  - **working forward:**



  - **new value is a (long) prefix of old value**

Adding & Removing Elements

- Can efficiently add & remove at end of array:

```
ArrayList<Integer> A = …;
A.remove(A.size() - 1);  // remove last element
```

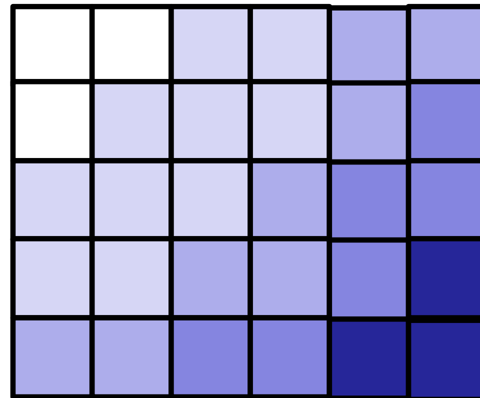
- How do we handle this in Floyd logic?
  - working backward:

```
{{ P(A[:len(A)-1]) }}
A.remove(A.size() - 1);
{{ P(A) }}
```

  - substitution through assignment $A = A_0[:len(A0)-1]$

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

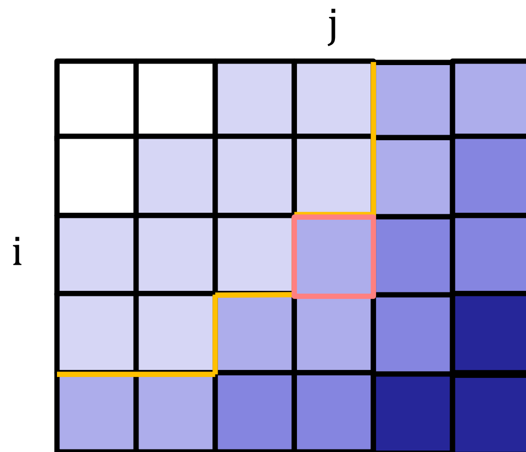
Given a sorted matrix M, with m rows and n cols, where every row and every column is sorted, find out whether a given number x is in the matrix

$_ < x$ $\quad$ $x \leq _$

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

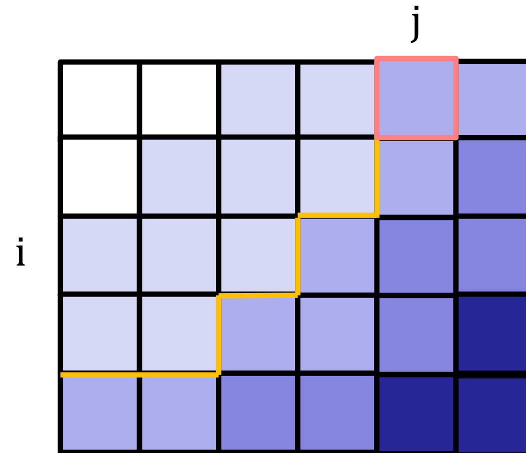
Sorted Matrix Search

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

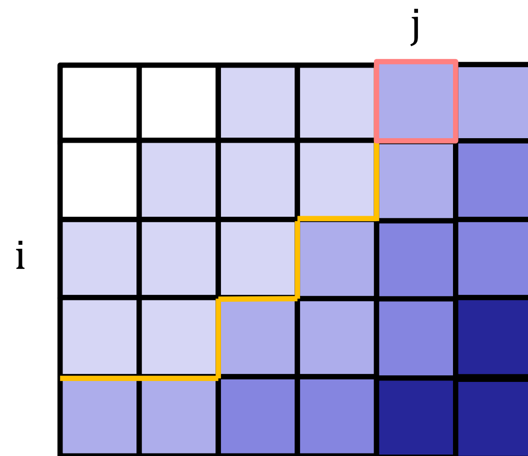


Need to *search* to find this location

Sorted Matrix Search

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

– will need a loop...



How do we find an invariant for that loop?

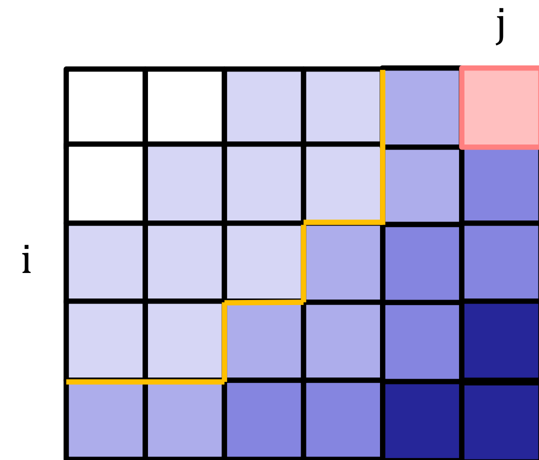
– try weakening this assertion (allow any j, not just smallest)

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

Sorted Matrix Search

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

```
int i = 0;
int j = ??
{{ Inv: x ≤ M[0, k] for any j ≤ k < n }}
while (??)
   ??
{{ Post: M[0, k] < x for any 0 ≤ k < j and x ≤ M[0, k] for any j ≤ k < n }}
```



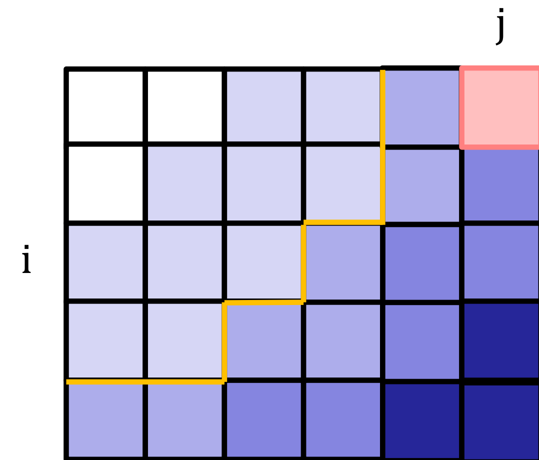
How do we set j to make Inv hold initially?

– range is empty when $j = n$

Sorted Matrix Search

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

```
int i = 0;
int j = n;
{{ Inv: x ≤ M[0, k] for any j ≤ k < n }}
while (??)
   ??
{{ Post: M[0, k] < x for any 0 ≤ k < j and x ≤ M[0, k] for any j ≤ 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

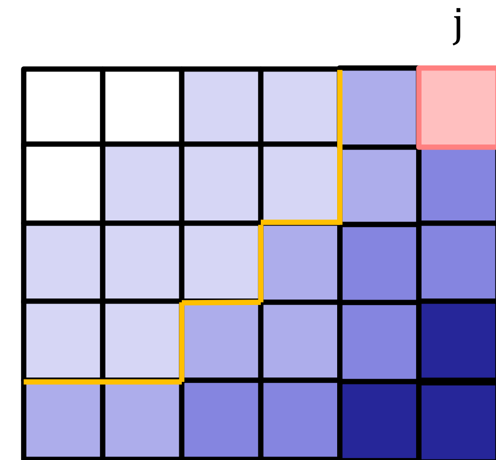
New Goal: find smallest j with $x \leq M[0, k]$ for any $j \leq k < n$

```
int i = 0;
int j = n;
```

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

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

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



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

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 ≤ 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 $x \leq M[0, j-1]$ }}

```
  ??
```

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

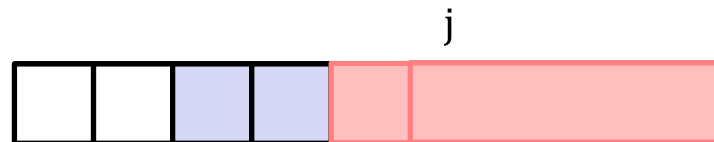
```
  j = j - 1;
```

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

```
}
```

Sorted Matrix Search

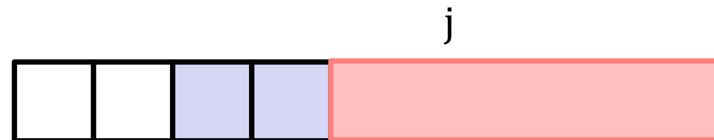
New Goal: find smallest j **with** $x \le M[0, \mathbf{k}]$ for any $j \le k < n$



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

??

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



Nothing is missing!

Sorted Matrix Search

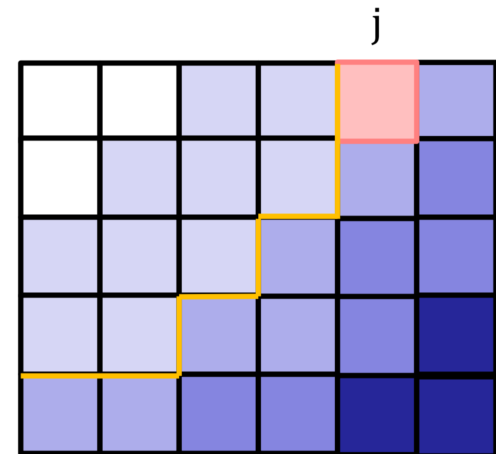
New Goal: find smallest j **with** $x \leq M[0, k]$ for any $j \leq k < n$

```
int i = 0;
int 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 $j \leq k < n$ }}

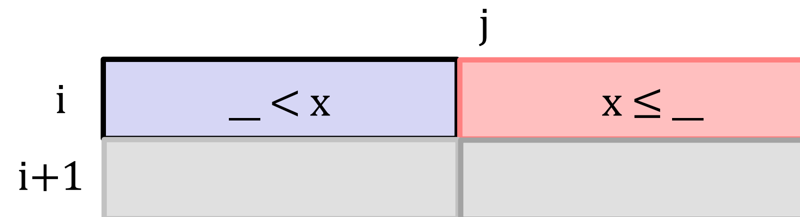


Can now check if $M[0, j] = x$

- **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 $i+1$

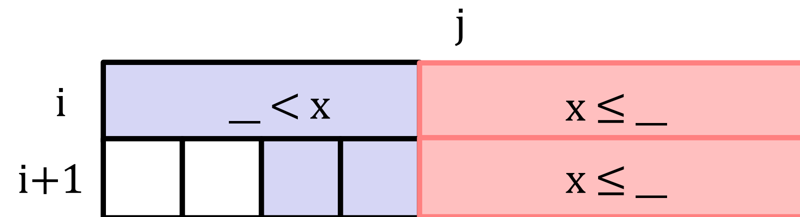


What does *vertical* sorting tell us about row $i+1$?

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

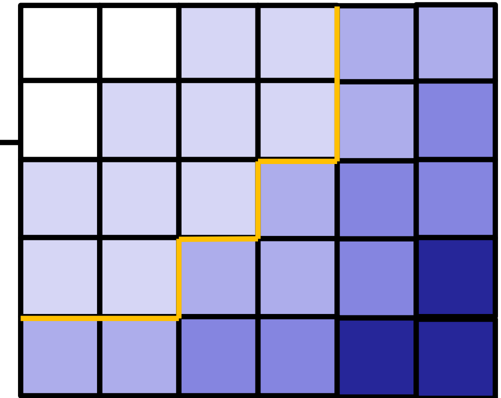
Sorted Matrix Search

Moving from row i to row i+1

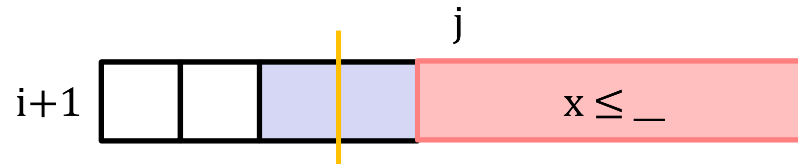


Next row looks like this

Sorted Matrix Search



Moving from row i to row i+1



How do we restore the invariant?

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

This is the same problem as before!

- move left until begining or $M[i+1, j-1] < x$ holds

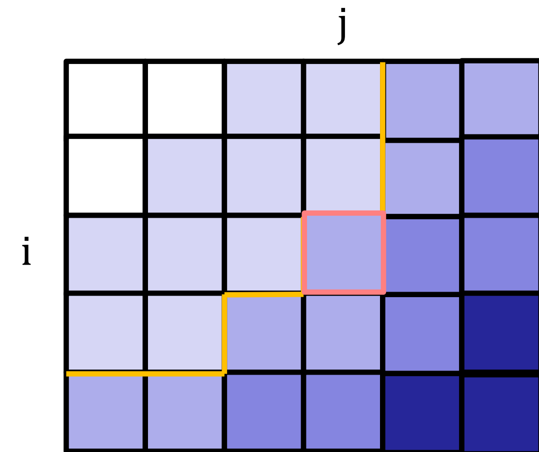
Sorted Matrix Search

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

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

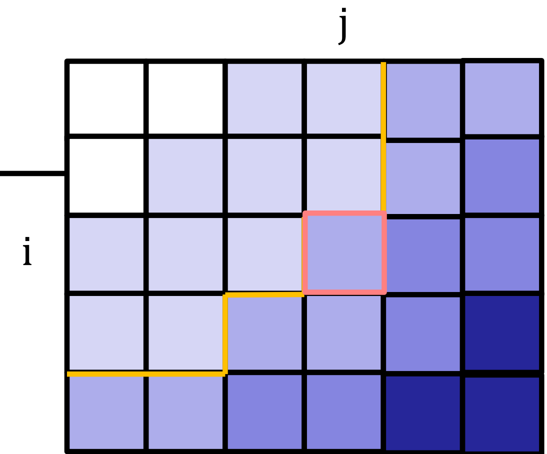
(M[i, k] $< x$ for any $0 \leq k < j$) and ($x \leq$ M[i, k] for any $j \leq k < n$) }}

```
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




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

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

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

```
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

Loop condition was also changed

```
int i = 0;
int 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;
```

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!

```
int i = 0;
int 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;
```